CARLA TIAKI UTSUNOMIYA

MÉTRICAS 00 APLICADAS A CÓDIGO OBJETO JAVA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática pelo Curso de Pós-Graduação em Informática, do Setor de Ciências Exatas da Universidade Federal do Paraná, em convênio com o Departamento de Informática da Universidade Estadual de Maringá.

Orientador: Prof. Dr. Márcio Eduardo Delamaro

CURITIBA

2003

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, da aluna *Carla Tiaki Utsunomiya*, avaliamos o trabalho intitulado, "*Métricas 00 Aplicadas a Código Objeto Java*", cuja defesa foi realizada no dia 29 de agosto de 2003, às dezesseis horas, no Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação da candidata. (Convênio número 279-00/UFPR de Pós-Graduação entre a UFPR e a UEM - ref. UEM número 1331/2000-UEM).

Curitiba, 29 de agosto de 2003.

Prof. Dr. Márcio Eduardo Delamaro
UNIVEM – Orientador

Prof. Dr. Edmundo Sérgio Spoto
UNIVEM – Membro Externo

Profª. Dra. Silvia Regina Vergílio
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

AGRADECIMENTOS

Primeiramente a Deus, pois sem Ele eu nada teria feito e por tudo o que Ele tem me concedido. A Deus toda a glória !!

À minha querida e amada família que possui um valor inestimável, por todo o amor demonstrado, apoio e compreensão.

Ao meu orientador o Prof. Dr. Márcio Eduardo Delamaro por ter me dado a oportunidade de trabalhar mediante a sua competência e pela sua compreensão e profissionalismo demonstrados.

Aos meus amigos, pelas orações e companheirismo, em especial à minha grande amiga Gláucia Petrucci.

Ao Prof. Josmar Mazucheli pela sua paciência e por seus valiosos auxílios.

Ao Prof. Edmundo Spoto por sua ajuda prestada desde o início deste Curso de Pós-Graduação.

Ao Departamento de Informática da Universidade Federal do Paraná, pela oportunidade.

Ao Auri Vincenzi da Universidade de São Paulo - São Carlos por todo o apoio dado.

À Universidade Estadual de Maringá, em especial ao Departamento de Informática, por disponibilizar todos os recursos necessários para a realização deste trabalho.

Aos professores do Departamento de Informática, especialmente aqueles que fizeram parte da comissão para a abertura do convênio do Curso de Pós-Gradução da Univeridade Federal do Paraná.

Aos funcionários do Departamento de Informática da Universidade Estadual de Maringá, em especial à Inês por sua dedicação e prestatividade.

À Profa. Rosângela pelas ajudas prestadas.

A todos aqueles que indiretamente participaram da realização deste trabalho.

DEDICATÓRIA

"Este trabalho é dedicado em especial ao meu grande amigo Jesus, aos meus familiares e preciosos amigos que fazem a vida valer a pena."

"Porque Deus amou o mundo de tal maneira que deu o seu filho único, para que todo aquele que nele acredita não morra, mas tenha a vida eterna."

— Jo 3:16 - Bíblia Sagrada —

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Na busca de melhorias no processo de desenvolvimento de software para a obtenção de um produto de qualidade, várias métricas têm sido propostas, com as quais pode-se gerenciar este processo e detectar falhas de projeto. As métricas de software auxiliam na coleta de informação, fornecendo dados qualitativos e quantitativos sobre o processo e o produto de software. Elas identificam onde os recursos são necessários, constituindo assim importante fonte de informação para a tomada de decisão. Métricas podem ser aplicadas em diversas fases do desenvolvimento e em diversos produtos intermediários como especificação de requisitos, projeto ou código fonte. Este trabalho mostra a factibilidade de coletar-se algumas métricas de software a partir do código objeto (bytecode) Java. Tal abordagem pode ser útil em atividades como: teste de programas que utilizam componentes de terceiros, re-engenharia e outras nas quais não se tenha acesso ao código fonte. As métricas coletadas a partir do bytecode Java foram aplicadas em um estudo de caso com dois sistemas onde procurou-se relacionar as métricas com a propensão a falhas.

Palavras Chaves: Métricas, Sistema Orientado a Objetos, Código Objeto e Probabilidade de Ocorrência de Falha.

ABSTRACT

Searching for improvements in the software development process and aiming at a product with high quality, several metrics, that help to manage the software process and to detect project flaws, have been proposed. Software metrics provide qualitative and quantitative about the software process and the software product. The identify where resources should be allocated, being an important source for decision making. They can be applied in any phase of the development and on different intermediary products as requirement specification, design or source code. This work shows the feasibility of collecting some software metrics directly from Java object programs (Java bytecode). Such an approach can be useful in activities like: testing of third party components, re-engineering and others activities where the source code is not available. This approach has been applied in a case study to two Java systems, trying to relate the metrics to the existence of faults.

Keywords: Metrics, Object-Oriented System, Source Object and Probability of Fault Detection.

CAPÍTULO 1

INTRODUÇÃO

O desenvolvimento de um sistema de software requer tempo e utilização de recursos. Desta forma, para a automação das atividades de desenvolvimento de um software de qualidade é fundamental a geração de informações precisas para se alocar recursos e tempo adequados no processo de desenvolvimento de software [46].

As métricas de software auxiliam este processo de coleta de informações fornecendo dados qualitativos e quantitativos sobre o processo e o produto de software. Elas identificam onde os recursos são necessários e são fontes cruciais de informações para a tomada de decisão [28]. Teste de sistemas pode ser citado como um exemplo de uma atividade que consome tempo e recursos. Aplicar-se teste e esforços de verificação iguais para todas as partes de um sistema de software acarreta um aumento nos custos e no tempo [46]. A capacidade de identificar antecipadamente módulos com propensão a falhas pode propiciar economia no desenvolvimento permitindo que os esforços de teste/verificação sejam concentrados nestes módulos [27]. Desta forma, torna-se essencial uma disponibilidade adequada das métricas de projeto para prever erros em módulos, tanto para uma melhora na qualidade do desenvolvimento do software quanto para a confiabilidade do produto final.

As métricas podem ser divididas em várias categorias que medem aspectos diferentes do programa, como por exemplo: métricas gerais, métricas de requisitos do software, métricas de projeto do software, métricas de código e métricas de teste. Cabe ao desenvolvedor considerar os aspectos que deseja medir em seu sistema. Neste trabalho, serão consideradas as métricas de projeto e as de código para análise e estudo.

Métricas de software existem desde a década de 60, como por exemplo, as medidas do tamanho de um produto em linhas de código ou o índice Fog [25]. São aplicadas em todas as fases do desenvolvimento e aos mais variados produtos deste processo, como

especificação, projeto ou código. Com o aparecimento e a popularização do paradigma de desenvolvimento orientado a objetos (OO), surgiram também diversas métricas que visam avaliar características relacionadas a aspectos particulares da OO, como herança ou composição.

As métricas OO podem ser coletadas a partir do projeto de classes ou do código fonte das mesmas. Entretanto, existem situações em que tais informações podem ser úteis ao engenheiro de software e para as quais não se dispõe do código fonte e, menos ainda, do projeto das classes. É o caso, por exemplo, do teste de programas baseados em componentes, em que parte do código pode ser produzido por terceiros e da qual não se tem acesso ao código fonte. Também é o caso da re-engenharia a partir do código objeto.

Programas objeto Java, ou mais precisamente, programas representados através de bytecode que possam ser executados pela Máquina Virtual Java [13](discutido na Seção 3.1), mantêm muitas das informações relativas às características de orientação a objetos que são utilizadas nas métricas OO e que, normalmente, são obtidas no código fonte.

Este trabalho propõe a aplicação das principais métricas OO, propostas e validadas na literatura, em um contexto ligeiramente diferente daquele em que são geralmente aplicadas. Em geral, as métricas OO são coletadas a partir do código fonte ou do projeto de classes do software. Neste trabalho, descreve-se como tais métricas podem ser obtidas a partir do código objeto Java e como elas foram implementadas numa ferramenta denominada JaBUTi. Como estudo de caso, foram utilizados dois sistemas produzidos em Java, nos quais as métricas foram coletadas a partir do bytecode. Procurou-se então relacionar as métricas com a propensão a falhas nos módulos do software. Uma classe foi considerada propensa a falha neste trabalho, se a mesma a mesma sofreu alguma modificação de código.

No próximo capítulo serão introduzidos os conceitos relacionados às métricas e como as mesmas podem ser avaliadas. No Capítulo 3 relata-se a análise estática do bytecode Java através da Ferramenta JaBUTi e como as métricas OO são coletadas a partir deste bytecode. No Capítulo 4, serão descritas as métricas implementadas na Ferramenta JaBUTi e como as mesmas foram implementadas através da classe **metrics.Metrics**. No

Capítulo 5 apresenta-se o estudo de caso onde as métricas são aplicadas sobre dois sistemas realizando-se a avaliação da co-relação entre os valores das métricas e a propensão a falhas de cada classe dos sistemas. Por fim, o Capítulo 6 relata os comentários finais sobre o trabalho.

CAPÍTULO 2

MÉTRICAS

Métricas de software existem de longa data. Várias delas foram propostas e utilizadas para medir características de programas "tradicionais", dentro do paradigma estruturado. É o caso, por exemplo, de medidas como Linhas de Código e Complexidade Ciclomática [20]. Com o surgimento do paradigma orientado a objetos (OO) e o aumento da utilização das técnicas relacionadas a este paradigma, cresceu a necessidade de se criarem novas métricas que abordassem as particularidades dos novos conceitos introduzidos com esta nova tecnologia, tais como classe, polimorfismo, herança e encapsulamento. Assim, várias métricas têm sido propostas na literatura de orientação a objetos com a finalidade de alcançar uma qualidade estrutural de programas e projetos orientados a objetos [11].

Estas métricas OO têm sido validadas empiricamente e aplicadas em vários sistemas OO existentes, como em [10, 11, 16, 17, 19, 22, 24, 34, 35, 37, 42, 44, 46]. Por exemplo, em [11,34,46] são relatados estudos de relacionamento entre projetos e medidas de código fonte em um conjunto de dados de sistemas de estudantes. Um outro estudo [35], descreve a validação de um conjunto de métricas OO em um sistema industrial. Chidamber *et al.* [44] descreve uma análise exploratória no qual investigam o relacionamento entre métricas OO e produtividade, esforços de manutenção e de projeto em 3 diferentes sistemas financeiros e Tang *et al.* [37] investiga o relacionamento entre um conjunto de métricas OO e falhas encontradas em 3 sistemas. Em [17], métricas OO são aplicadas para garantir a manutenibilidade do sistema.

Neste capítulo, serão introduzidos os conceitos relacionado às métricas. Primeiramente, na Seção 2.1, abordaremos as métricas de software em geral, logo após na Seção 2.2, uma classificação das métricas de qualidade de software. Na Seção 2.3, as principais métricas de código e na Seção 2.4 algumas das métricas OO são abordadas. A avaliação das métricas OO efetuada em grande parte dos trabalhos da área é discutida na Seção 2.5,

juntamente com alguns dos resultados obtidos desta avaliação em alguns trabalhos. O capítulo é finalizado na Seção 2.6 com um exemplo de aplicação das métricas discutidas.

2.1 MÉTRICAS DE SOFTWARE

A medição é um recurso utilizado há vários séculos principalmente na ciência e na engenharia. Galileu escréveu há 500 anos atrás "O que não for mensurável faça mensurável." ("*What is not measurable make measurable.*")

Tom DeMarco também disse anos atrás algo a que tem se dado real importância nos dias de hoje: "Não se pode controlar o que não se pode medir." ("*You can not manage what you do not measure.*") [7]

Fenton e Pfleeger em 1997 [43] definiram o ato de se medir (mensuração) como sendo o processo pelo qual números ou símbolos são associados a atributos de entidades do mundo real, de modo a descrevê-los segundo regras claramente definidas. Assim, as métricas são utilizadas para se obter dados quantitativos ou qualitativos de entidades do mundo real.

As métricas de software são utilizadas nas empresas desenvolvedoras para alcançar maior produtividade, através da padronização do processo, com menor custo e maior qualidade.

Desta forma, as métricas de software têm o objetivo de [39]:

- Melhor compreender a qualidade do produto:

  Os dados resultantes das métricas servem como base para estimativas possibilitando a compreensão do estado de qualidade do software, tanto do produto intermediário, quanto do produto final. As métricas podem ser aplicadas no decorrer do projeto (produto em desenvolvimento), por exemplo, para indicar uma possível propensão a falhas. Também podem ser aplicadas no produto final, por exemplo, verificando a qualidade de um software através da consistência e completeza dos requisitos. Assim, as métricas contribuem para definir os requisitos de qualidade do software e medir, controlar e melhorar a qualidade do produto. Compreendendo a qualidade do produto, as métricas ajudam tanto na tomada de decisões quanto na liberação

ou aceitação do mesmo.

- Planejar e avaliar a efetividade do processo de software:

  Com a aplicação das métricas é possível planejar o processo de desenvolvimento do software identificando-se a quantidade de esforço, custo e tempo necessários para a conclusão do projeto. De acordo com o planejamento é possível acompanhar o progresso do projeto e determinar se o processo de desenvolvimento está sendo executado com desempenho apropriado. Assim, as melhores práticas podem ser validadas experimentalmente ao se medir os esforços, recursos (pessoas envolvidas e métodos e ferramentas utilizados) e tempo gastos em cada fase do processo de desenvolvimento de software. Medir a efetividade do processo de desenvolvimento evita gastos excessivos e contribui para uma melhora na qualidade de trabalho no decorrer do projeto.

2.2 TIPOS DE MÉTRICAS

Segundo a ISO/IEC 9126 [1], as métricas de qualidade de software podem ser divididas em três categorias: métricas internas, métricas externas e métricas de qualidade em uso. Desta forma, a qualidade do software poderia ser medida por atributos internos (medidas estáticas de produtos intermediários), atributos externos (medidas do comportamento do código quando executado) ou atributos de qualidade em uso (medir se o produto produz o efeito requerido no seu contexto particular de uso).

É visível a correlação entre estes três tipos de medidas, uma vez que atributos internos apropriados do software nos leva a alcançar o comportamento externo requerido, que consequentemente nos leva a uma qualidade em uso [1]. As métricas internas e externas medem o produto em si e as métricas de qualidade em uso indicam o quanto o software atende aos requisitos definidos pelo usuário e à necessidade de qualidade do mesmo.

As métricas internas visam a medir os atributos do software durante o seu desenvolvimento. São aplicadas durante a revisão de projeto e a revisão de código. Neste ponto, o software ainda não é executável. Os objetivos desta avaliação de qualidade interna são:

verificar se os requisitos de qualidade interna (requisitos para a fase de projeto do software para alcançar a qualidade externa requerida) estão sendo satisfeitos (verificação) e prever a qualidade final do produto em desenvolvimento (validação). A medição é feita em especificações ou código fonte.

Já as métricas externas medem os atributos do sistema. São aplicadas na fase de testes. O objetivo desta avaliação é verificar se todos os requisitos de qualidade estão sendo satisfeitos (validação).

As métricas em uso visam a medir a satisfação do usuário com o produto final. São aplicadas após a liberação do software, quando os usuários o utilizam em seu ambiente de trabalho com dados reais. É possível identificar algumas necessidades que não foram explicitadas. Métodos de medição: feedback dos usuários através de questionários, observação do comportamento do usuário ou outras medições locais.

Alguns requisitos para a qualidade do software são incluídos para assegurar critérios para a qualidade interna, externa e em uso.

Requisitos para a qualidade interna e externa ou características de métricas externas e internas são: funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade. Cada uma destas características possui subcaracterísticas próprias (desdobramentos) que ao serem medidas contribuem para quantificar e assim compreender a característica em questão. A Tabela 2.1 descreve resumidamente as características e suas respectivas subcaracterísticas com os seus significados das métricas externas e internas.

Requisitos para a qualidade ou características referentes às métricas de qualidade em uso são: eficácia, produtividade, segurança e satisfação. Estas características cobrem todas as características da qualidade externa e qualidade interna. A Tabela 2.2 descreve resumidamente as características e os significados das métricas de qualidade em uso.

A Figura 2.1 demonstra a qualidade no ciclo de vida do software [1]:

De acordo com a Figura 2.1, os requisitos e os produtos "caminham em direções contrárias". Os requisitos são definidos de acordo com as necessidades do usuário, que contribuem para a especificação dos requisitos de qualidade externa (do sistema), que por sua vez contribuem para a especificação dos requisitos de qualidade interna (do software).

Tabela 2.1: RESUMO DAS CARACTERÍSTICAS DAS MÉTRICAS INTERNAS E EXTERNAS

Característica	**Subcaracterística**	**Significado** (capacidade)
Funcionalidade	Adequação	de prover um conjunto apropriado de funções para tarefas e objetivos especificados
	Acurácia	de prover resultados corretos ou com o grau de precisão necessário
	Interoperabilidade	de interagir com um ou mais sistemas especificados
Confiabilidade	Maturidade	de evitar deficiências decorrentes de falhas no software.
	Tolerância a falhas	de manter um nível de desempenho especificado em casos de falhas ou de violação da interface especificada
	Recuperabilidade	de restabelecer seu nível de desempenho e recuperar os dados diretamente afetados no caso de ter uma deficiência
	Conformidade	de estar de acordo com normas, convenções ou regulamentações relativos a confiabilidade
Usabilidade	Inteligibilidade	de permitir ao usuário compreender se o software se aplica às suas necessidades e como ele pode ser usado para determinadas tarefas e condições de uso
	Apreensibilidade	de permitir ao usuário aprender sua aplicação
	Operacionalidade	de possibilitar o usuário a operá-lo e controlá-lo
	Atratividade	de ser atrativo ao usuário
	Conformidade	de estar de acordo com normas, convenções, guias de estilo ou regulamentações relativas a usabilidade
Eficiência	Comportamento com relação ao tempo	de fornecer tempo de resposta e tempo de processamento apropriados e taxas de transferência quando executando suas funções, sob condições estabelecidas
	Utilização de recursos	de usar quantidade e tipos de recursos apropriados quando o software executa suas funções sob condições estabelecidas
	Conformidade com eficiência	de estar de acordo com normas e convenções relativas a eficiência
Manutenibilidade	Analisabilidade	de permitir o diagnóstico de deficiências ou causas de falhas no software, ou a identificação de partes a serem modificadas
	Modificabilidade	de permitir que a modificação especificada seja implementada
	Estabilidade	de minimizar efeitos inesperados de modificações de software
	Testabilidade	de permitir o software modificado ser validado
	Conformidade com manutenibilidade	de estar de acordo com normas ou convenções relativas a manutenibilidade
Portabilidade	Adaptabilidade	de ser adaptado para diferentes ambientes especificados sem necessidade de aplicação de outras ações ou meios além daqueles fornecidos para essa finalidade pelo software considerado
	Capacidade para ser instalado	para ser instalado em um ambiente especificado
	Coexistência	para coexistir com outros software independentes em um ambiente comum compartilhando recursos comuns
	Capacidade para substituir	para ser usado em substituição de outro produto de software especificado para o mesmo propósito no mesmo ambiente
	Conformidade com portabilidade	para aderir a normas ou convenções relativas a portabilidade

Tabela 2.2: RESUMO DAS CARACTERÍSTICAS REFERENTES ÀS MÉTRICAS DE QUALIDADE EM USO

Característica	**Significado** (capacidade)
Eficácia	de permitir aos usuários atingir metas especificadas com acurácia e completude em um contexto de uso especificado
Produtividade	de permitir ao usuário usar a quantidade de recursos apropriados em relação à eficácia atingida quando o produto de software é utilizado em um contexto de uso especificado. Os recursos podem incluir tempo, esforço, materiais ou custos financeiros.
Segurança	de atingir níveis aceitáveis de riscos de danos para pessoas, negócios, software, propriedades ou ambiente em um contexto de uso especificado. Os riscos usualmente são resultados das deficiências na funcionalidade (incluindo segurança), confiabilidade, usabilidade e manutenibilidade.
Satisfação	de satisfazer os usuários em um contexto de uso especificado. É a resposta do usuário à interação com o produto e inclui atitudes pelo uso do produto.

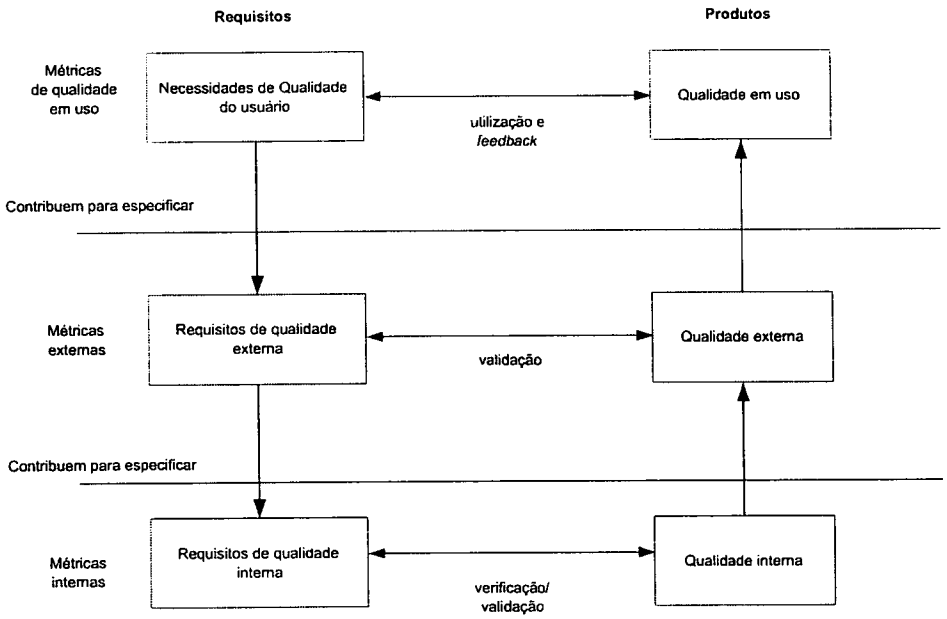


Figura 2.1: QUALIDADE NO CICLO DE VIDA DE SOFTWARE

Para o produto, a qualidade interna (do software) contribui para se chegar na qualidade externa (do sistema) que contribui para se chegar na qualidade em uso para o usuário. Em cada fase, os requisitos são utilizados para verificar e validar a qualidade do produto.

Para se ter uma idéia mais concreta dos três tipos de métricas comentadas até aqui, são apresentados três exemplos, um para cada tipo, extraídos de [1].

Característica: confiabilidade

Subcaracterística: tolerância a falhas

Métrica Interna

Nome da métrica: capacidade de evitar falhas

Propósito: determinar o número de falhas previstas e evitadas no código

Fórmula: (♯ falhas previstas no projeto / ♯ falhas possíveis)

Interpretação: $0 \leq x \leq 1$; quanto mais próximo de 1, melhor

Entradas: relatório de revisão, especificação de requisitos

Métrica Externa

Nome da métrica: capacidade de evitar falhas

Propósito: determinar o controle de ocorrência de falhas

Fórmula: (# falhas evitadas / # casos de teste)

Interpretação: $0 \leq x \leq 1$; quanto mais próximo de 1, melhor

Entradas: relatório de teste e de operação

Característica: eficácia

Métrica Qualidade em Uso

Nome da métrica: tarefas completadas

Propósito: determinar a proporção de tarefas completadas

Fórmula: (♯ tarefas completadas / ♯ tarefas tentadas)

Interpretação: $0 \leq x \leq 1$; quanto mais próximo de 1, melhor

Entradas: relatório de operação, registro de histórico de uso

Grande parte da utilização das métricas nas mais diversas áreas, resultam em medi-

das quantitativas absolutas. Por exemplo, na física, as métricas resultam em medidas absolutas de voltagem, velocidade ou temperatura. Porém, ao lidarmos com métricas de software, os resultados são medidas que tentam associar as métricas com alguma indicação de qualidade do produto ou processo do software [39].

As métricas de software podem ainda ser divididas em duas outras categorias, como as medições no mundo físico: métricas diretas e métricas indiretas. No mundo físico, as medidas diretas se caracterizam por serem coletadas direta e facilmente, como por exemplo, o comprimento de um parafuso e as medidas indiretas se caracterizam por serem coletadas indiretamente e mais dificilmente, pois referem-se a uma característica "abstrata" do objeto medido, como por exemplo, a "qualidade" dos parafusos produzidos, medida pela contagem dos parafusos rejeitados [39]. Da mesma forma, as métricas de software podem ser diretas ou indiretas. O tempo, custo e esforço aplicados no processo de desenvolvimento e o número de linhas de código produzido podem ser considerados como medidas diretas e características de qualidade. Funcionalidade do software e eficiência como medidas indiretas [39].

As métricas podem também ser classificadas de acordo com o objeto de aplicação. A especificação de um projeto, os relatórios de testes, os documentos de manutenção são considerados diferentes objetos de aplicação. Por exemplo, a métrica funcional Pontos por Função (PF - *Function Point*) é aplicada na especificação do software.

As métricas funcionais visam a medir a funcionalidade ou utilidade do software e são consideradas métricas indiretas [39]. Inicialmente proposta na década de 70, por pesquisadores da IBM, que procuravam determinar escalas para medir a produtividade de programação e descobriram que poderiam basear a avaliação de um software medindo o valor das funções executadas pelos programas, em vez de utilizar como base o volume ou a complexidade do código dos programas [4].

Seguindo estas pesquisas, em 1979, Allan Albrecht [30] criou a técnica de avaliação FP que é utilizada para comparar produtividade (esforço de programação) de diferentes equipes, técnicas e tecnologias. A métrica qualifica a performance do produto pela visão externa do usuário, independente da linguagem de programação usada e ajuda o usuário

a melhor avaliar a qualidade do software. A métrica também auxilia na estimação do tamanho do sistema, do custo de software e a taxa de manutenção de software [3].

A métrica FP é independente da linguagem de programação e se baseia em dados que são conhecidos no começo da evolução do projeto, sendo utilizada como estimativa [39].

Além da métrica FP, existem várias outras métricas que são utilizadas para a medição. A seguir serão discutidas algumas das principais métricas tradicionais e OO de software propostas na literatura. Esse sumário baseia-se em [11, 15, 26, 39].

2.3 MÉTRICAS DE CÓDIGO

As principais métricas tradicionais que são aplicadas nos programas ou ainda, nos projetos, serão comentadas com maiores detalhes. Estas métricas foram divididas em: métricas de tamanho e de complexidade.

2.3.1 Métricas de Tamanho

As métricas de tamanho são medidas diretas do software e do processo por meio do qual ele é desenvolvido [39]. Elas provocam controvérsias e não são universalmente aceitas como a melhor maneira de se medir o processo de desenvolvimento de software [31]. Grande parte da discussão está relacionada a considerar a contagem de linhas de código como forma de medida.

A contagem de linhas de código de um software (também conhecida como LOC - *Line of Code*) foi a primeira forma encontrada para medir o tamanho de um sistema [29]. A métrica mede o "tamanho" do software que tem sido produzido, e é calculada contando-se o número de linhas de código do projeto [39]. Algumas das desvantagens da métrica são: (1) alguns desenvolvedores acreditam que as medidas LOC são dependentes da linguagem de programação, uma vez que cada linguagem possui características específicas e ao se comparar dados de medidas de projetos que não utilizaram a mesma linguagem de programação resultados insatisfatórios são gerados, (2) pelo fato desta técnica ser aplicável apenas após a codificação dos programas, a mesma não permite estimativas em proje-

tos [23] e (3) através dela não se pode medir a complexidade de um sistema, uma vez que maior LOC não é equivalente a um sistema bem projetado.

Tainbém existem diferentes opiniões sobre o que considerar como uma linha de código. Deve-se definir se linhas de comentário, linhas em branco e início e final de blocos serão considerados como linhas de código.

Exemplos de utilização da medida LOC:

- o esforço gasto em determinada fase do processo de desenvolvimento de software pode ser medido pela média da divisão do número de LOC produzidos nesta fase pelo número de pessoas que trabalharam nesta fase do projeto;
- a qualidade do software pode ser medida calculando-se a média do número de falhas encontradas pelo LOC.

2.3.2 Métrica de Complexidade

As métricas de complexidade visam a medir a complexidade do sistema. Dentre uma das importantes métricas de complexidade, a métrica de Complexidade Ciclomática de McCabe (CC) é utilizada para se avaliar a complexidade de um algoritmo em um método [5]. A métrica provê uma medida quantitativa da dificuldade de teste e uma indicação da confiabilidade final [39]. A dificuldade de teste é medida através do número de casos de testes (CC) que devem ser executados para garantir a cobertura de todas as declarações do programa [39].

A complexidade ciclomática é baseada nos grafos de fluxo de controle das unidades do sistema. Um grafo de fluxo de controle descreve a estrutura lógica de uma unidade do sistema. Uma unidade corresponde a uma função ou subrotina que tem uma entrada simples e um ponto de saída e é capaz de ser usado como um componente de projeto através de um mecanismo de chamada e retorno [20].

Os grafos de fluxo consistem de nós e ramos, onde os nós representam comandos ou expressões e os ramos representam a transferência de controle entre estes nós.

A métrica pode ser calculada das seguintes formas [39]:

- O número de regiões do grafo de fluxo corresponde à Complexidade Ciclomática;
- A Complexidade Ciclomática, V(G), para o grafo de fluxo G é definida como:

$$V(G) = E - N + 2$$

onde V(G) mede os caminhos linearmente independentes encontrados no grafo, E é o número de ramos do grafo de fluxo e N o número de nós.

- A Complexidade Ciclomática, V(G), para o grafo de fluxo G é definida como:

$$V(G) = P + 1$$

onde, P é o número de nós predicados contidos no grafo de fluxo G.

Outra métrica de complexidade é a métrica de Halstead. Em 1972, Maurice Halstead, iniciou estudos sobre algoritmos tentando testar empiricamente a hipótese de que os operadores (comandos e palavras reservadas) e os operandos (itens de dados) em um programa deveriam se relacionar com a quantidade de erros no algoritmo [26].

Com o sucesso dos estudos, em 1977 foi construído um sistema que registra o número de operadores e operandos utilizados para cada programa, permitindo o cálculo do tamanho do programa e o esforço de programação, independente da linguagem de programação. As desvantagens deste sistema é que o mesmo é baseado na sintaxe dos programas e não considera o seu conteúdo, o seu processo é incompreensível ao usuário final e só pode ser aplicado após a codificação dos programas, não permitindo estimativas em projetos (assim como LOC) [3].

2.4 MÉTRICAS ORIENTADAS A OBJETOS

Muitas das métricas de código propostas podem ser aplicadas em programas OO, mas as particularidades dos mesmos devem ser consideradas. As métricas de código propostas

não são apropriadas para medir características de qualidade OO como, por exemplo: herança, abstração e encapsulamento. Assim, as métricas orientadas a objetos foram criadas pela necessidade de se medir tais características.

As métricas para sistemas OO têm como função a obtenção das características numéricas das estruturas internas do software, que refletem a complexidade de cada componente individual, como métodos e classes; e na complexidade externa, que mede as interações entre as entidades, tal como acoplamento e herança [5]. Elas podem ser aplicadas no código ou no projeto.

Dentre as principais métricas existentes destacam-se dois grupos, conhecidos como as métricas CK (Chidamber e Kemerer) e as métricas LK (Lorenz e Kidd):

Chidamber e Kemerer (CK), 1994 [11] : *Weighted Methods per Class* (WMC), *Depth of Inheritance Tree* (DIT), *Number of Children* (NOC), *Response for a class* (RFC), *Coupling Between Objects* (CBO) e *Lack of Cohesion in Methods* (LCOM);

Lorenz e Kidd (LK), 1994 [15] : *Average Method Size* (AMZ), *Number of Public Instance Methods in a Class* (NPIM), *Number of Class Methods in a Class* (NCM), *Use of Multiple Inheritance* (UMI), *Number of Methods Overridden by a Subclass* (NMOS), *Number of Methods Inherited by a Subclass* (NMIS), *Number of Methods Added by a Subclass* (NMAS), *Number of Instance Variables in a Class* (NIV) e *Average Number of Parameters per Method* (ANPM).

Apesar destas métricas serem mais recentes, algumas pesquisas têm sido efetuadas na área para validá-las empiricamente (verificação de que uma medida é útil em um ambiente de desenvolvimento particular [5]) procurando estabelecer relacionamentos das métricas com algum aspecto de qualidade do software ou com seu processo de desenvolvimento. Na seleção de um conjunto de métricas para sistemas OO deve-se avaliar construções básicas do projeto, como métodos, classes, acoplamento e herança e devem ser escolhidas métricas que não dependam do ambiente de implementação e que sejam fáceis de se coletar [5].

Serão detalhadas as conhecidas métricas de Shyam Chidamber e Chris Kemerer (Subseção 2.4.1) e as métricas de Lorenz e Kidd (Tabela 2.3). As métricas de CK analisam e

descrevem as classes de um sistema orientado a objetos, pois os mesmos acreditam que o foco essencial de um sistema deste tipo está em seu projeto de classes.

Em [15], Lorenz e Kidd dividiram as métricas em: *project metrics* (métricas de projeto) e *design metrics* (métricas de desenho). As métricas de projeto medem aspectos dinâmicos do sistema em um nível mais alto de abstração, para predizer esforço, como por exemplo, em certo ponto do ciclo de vida do software verificar o quanto do projeto já foi executado e o quanto falta fazer. Já as métricas de desenho medem aspectos estáticos do software buscando medir a qualidade do software que tem sido desenvolvido, também em certo ponto do ciclo de desenvolvimento [15]. Estas últimas são mais específicas e levam a examinar e a melhorar a qualidade de certas partes do software.

As 9 métricas de projeto propostas foram divididas em: tamanho da aplicação, tamanho da alocação de pessoas e programação determinada.

A Tabela 2.3 relaciona as 32 métricas de desenho de LK com uma descrição sucinta de cada uma delas. Estas métricas foram divididas em: tamanho do método (NMS, M, NS, LOCM e AMZ), método interno (MC e SMS), tamanho da classe (NPIM, NIM, AIMC, NIV, AIVP, NCM e NCV), herança de classe (CHNL, NAC e UMI), herança de método (NMOS, NMIS, NMAS e SI), classe interna (CCoh, GU, ANPM, UFF, PFOC, ANCLM, ANCM e NPR) e classe externa (CCou, NTCR e NCMTA).

Tabela 2.3: RESUMO DE ALGUMAS DAS MÉTRICAS DE LK

Métrica	**Descrição**
Número de mensagens enviadas (*Number of message sends* - NMS)	mede o número de mensagens enviadas ao método, separado pelo tipo de mensagem.
Significado (*Meaning* - M)	quantifica o tamanho do método considerando outros fatores que LOC não considera, como o estilo do código.
Número de Declarações do método (*Number of Statements* - NS)	conta o número de declarações. Uma declaração é dependente da linguagem.
Linhas de Código do método (*Lines Of Code* - LOCM)	mede o número de linhas físicas de código ativo no método. [15]

Tabela 2.3: CONTINUAÇÃO DO RESUMO DE ALGUMAS DAS MÉTRICAS DE LK

Tamanho médio do método (*Average method size* - AMZ)	calculada pela divisão entre a soma da métrica LOCM dos métodos da classe pelo número de métodos na classe (soma dos métodos instância e classe). Segundo os autores um valor da métrica alto é um problema (provavelmente mais código orientado a função está sendo escrito do que código OO (projeto ruim)) [15].
Complexidade do Método (*Method Complexity* - MC)	número e tipos de mensagens enviadas em um método.
Strings de mensagens enviadas (*Strings of Message sends* - SMS)	útil para linguagem como Smaltalk, no qual mensagens podem ser enfileiradas juntas. Pode relacionar strings de mensagens enviadas a indicadores de recuperação de erro.
Número de métodos de instância públicas na classe (*Number of public instance methods in a class* - NPIM)	conta o número de métodos de instância públicas na classe. Mostra a quantidade de responsabilidade da classe.
Número de métodos de instância na classe (*Number of instance methods in a class* - NIM)	conta todos os métodos públicos, protegidos e privados definidos para uma instância de uma classe.
Número médio de métodos de instância por classe (*Average number of instance methods per class* - AIMC)	calculada através da divisão entre a somatória da métrica NIM de cada uma das classes do sistema pelo número total de classes. Também indica a quantidade de responsabilidade da classe no sistema.
Número de variáveis de instância na classe (*Number of instance variables in a class* - NIV)	o número de variáveis de instância na classe incluem as variáveis *public*, *private* e *protected* disponíveis para as instâncias.
Número médio de variáveis de instância por classe (*Average number of instance variables per class* - AIVC)	calculada através da divisão entre a somatória da métrica NIV de cada uma das classes do sistema pelo número total de classes. Segundo os autores, um valor alto para a métrica pode indicar que as classes estão fazendo mais do que elas deveriam fazer, se relacionando com vários outros objetos no sistema.

Tabela 2.3: CONTINUAÇÃO DO RESUMO DE ALGUMAS DAS MÉTRICAS DE LK

Número de métodos classe na classe (*Number of class methods in a class* - NCM)	número de métodos *static* na classe. Provê serviços e estado de dados que são globais às suas instâncias. Pode indicar projeto pobre se serviços que são melhor tratados por instâncias individuais são tratados pela própria classe. Assim, segundo os autores, o ideal é que NCM seja menor que NIM.
Número de variáveis classe na classe (*Number of class variables in a class* - NCV)	número de variáveis *static* na classe. Estão localizadas globalmente provendo objetos comuns para todas as instâncias de uma classe. Da mesma forma, o ideal é que NCV seja menor que NIV.
Nível de aninhamento da hierarquia de classe (*Class hierarchy nesting level* - CHNL)	maior nível de aninhamento das classes ou a profundidade do grafo na hierarquia de classes. Um alto valor indica que subclasses não são especialização de todas as superclasses, dificultando assim, o teste da classe (problema de projeto).
Número de classes abstratas (*Number of abstract classes* - NAC)	uma classe abstrata existe para facilitar o reuso de métodos e o estado de dados sobre suas subclasses. Uma superclasse não deve ser abstrata e experiências mostram que poucas classes abstratas resultam em projetos com sucesso e ao mesmo tempo a sua utilização indica uso de herança com sucesso.
Uso de herança múltipla (*Use of multiple inheritance* - UMI)	métrica boleana que indica a presença ou não de herança múltipla. Os autores recomendam a não utilização da herança múltipla, considerando-o uma anormalidade, pois o seu uso pode introduzir alguns problemas.
Número de métodos sobrescritos na subclasse (*Number of methods overridden by a subclass* - NMOS)	uma subclasse pode definir um método com o mesmo nome de um método de sua superclasse (sobrescrever). Uma mensagem a este método causará a execução do método da subclasse. Um valor alto desta métrica indica problemas de projeto, desde que uma subclasse deve ser uma especialização de sua superclasse.
Número de métodos herdados pela subclasse (*Number of methods inherited by a subclass* - NMIS)	subclasses naturalmente herdam comportamento na forma de métodos e estado de dados na forma de variáveis de instância de suas superclasses. Indica a força da subclasse através da especialização.

Tabela 2.3: CONTINUAÇÃO DO RESUMO DE ALGUMAS DAS MÉTRICAS DE LK

Número de métodos adicionados pela subclasse (*Number of methods added by a subclass* - NMAS)	número de novos métodos adicionados pelas subclasses. Subclasses definem novos métodos estendendo o comportamento de suas superclasses. O número de novos métodos deve diminuir ao se mover para baixo na árvore de hierarquia de herança.
Índice de Especialização (*Specialization index* - SI)	especialização pura implica em adicionar mais comportamento enquanto utiliza-se o comportamento existente. Na prática, especialização através de subclasses inclui adicionar novos métodos, adicionar comportamento para métodos existentes, sobrescrever métodos com novo comportamento e excluir métodos sobrescrevendo-os sem nehum comportamento. Calculado pela divisão entre o resultado da multiplicação de NMOS e CHNL pelo número total de métodos. Um alto valor da métrica indica que existem classes na hierarquia de classes que não estão abstraindo adequadamente suas superclasses [39].
Coesão da Classe (*Class Cohesion* - CCoh)	mensagens de conexões dentro da classe e o uso de variáveis de instância são formas de coesão (o que é desejável). Mede a inter-relação entre os métodos da classe e o padrão de uso das variáveis usadas pelos métodos da classe.
Uso global (*Global usage* - *GU*)	Quantidade de objetos disponíveis no sistema. Segundo os autores, um valor alto indica projeto pobre, uma vez que encoraja acoplamento desnecessário.
Número médio de parâmetros por método (*Average number of parameters per method* - ANPM)	parâmetros requerem mais esforços dos clientes, o que implica no estilo do projeto. Um alto valor da métrica coloca uma responsabilidade maior no cliente.
Uso de funções *friend* (*Use of friend functions* - UFF)	mede o uso de funções *friend* que rompem o encapsulamento.
Porcentagem de código orientado a funções (*Percentage of function-oriented code* - PFOC)	quantidade de código escrito fora das classes. Código não OO normalmente é visto como um problema de projeto, indicando uma regressão para código orientado a função.

Tabela 2.3: CONTINUAÇÃO DO RESUMO DE ALGUMAS DAS MÉTRICAS DE LK

Número médio de linhas comentadas por método (*Average number of comment lines per method* - ANCLM)	quantidade de comentários incluídos nos métodos. Indica se comentários suficientes estão sendo escritos. Comentários de convenções de projeto ou ferramenta de inserção automática (como data) não devem ser contados como comentários de explicação.
Número médio de métodos comentados (*Average number of commented methods* - ANCM)	indica o conjunto de todos os comentários incluídos nos métodos. Da mesma forma que ANCLM, indica se comentários suficientes estão sendo escritos.
Número de problemas reportados por classe ou contrato (*Number of problem reports per class or contract* - NPR)	uma classe com um grande número de problemas reportados é forte candidata para ser reescrita, indicando projeto pobre.
Acoplamento da classe (*Class coupling* - CCou)	avalia as conexões entre as classes através das mensagens existentes entre elas. O acoplamento é determinado pelo nível de dependências entre as classes.
Número de vezes que a classe é reutilizada (*Number of times a class is reused* - NTCR)	número de referências à uma classe. Existem algumas formas para contar-se o reuso, incluindo o reuso caixa preta e caixa branca.
Número de classes ou métodos jogados fora (*Number of classes/methods thrown away* - NCMTA)	número de classes e/ou métodos descartados durante o desenvolvimento. Segundo o autor, para desenvolver soluções elegantes é preciso algumas repetições nos projetos (o último projeto é melhor que o projeto anterior).

2.4.1 Descrição das Métricas de CK

Em [11] Chidamber e Kemerer propõem um conjunto de métricas para melhorar o processo de desenvolvimento de software de projetos OO. A criação de seis métricas de projeto OO é, segundo os autores, devido à demanda de desenvolvimento de software OO e o foco no melhoramento do processo ter aumentado a demanda para métricas de software, com os quais gerencia-se o processo. Estas métricas são baseadas na teoria de medição e refletem pontos de vistas de experientes desenvolvedores de software OO.

Foram coletadas amostras empíricas destas métricas em 2 projetos comerciais diferentes para demonstrar-se como as mesmas podem ser utilizadas para melhorar o processo. As seis métricas de CK são descritas logo abaixo.

2.4.1.1 Número de Filhos (*Number of Children* - NOC)

Medida associada à classe. Número de subclasses imediatas subordinadas à classe na árvore de hierarquia de herança. Ela então é considerada uma métrica de herança [11], pois como a herança promove o reuso, quanto maior o número de filhos de uma classe, maior deve ser o reuso. Indica, também, o nível de teste requerido, ou, quanto mais filhos (subclasses) uma classe possui, mais complexa é e mais testes requer. É, também, um indicativo dos esforços gastos de manutenção, uma vez que uma classe com mais filhos tende a ter a sua manutenção mais delicada, visto que, uma mudança nela pode afetar várias outras subclasses.

2.4.1.2 Profundidade da Árvore de Herança (*Depth of Inheritance Tree* - DIT)

Métrica de herança que mede o nível máximo da hierarquia de herança de uma classe. É o maior caminho da classe à raiz na árvore de hierarquia de herança. Assume-se que quanto mais profunda uma classe se encontra na hierarquia de herança, mais complexa ela é e mais difícil é prever o seu comportamento, pelo fato de herdar mais definições, métodos, entre outros, de seus antecessores.

Outro ponto a se considerar é que quanto maior a profundidade de uma classe na hierarquia, mais difícil é a sua manutenção, mais provável é que ela apresente falhas.

Indica o potencial para reuso (quanto mais profunda a classe na hierarquia, maior o potencial de reuso dos métodos herdados) e a complexidade do projeto (quanto maior a árvore de herança, mais complexo é o projeto).

2.4.1.3 Número ponderado de métodos por classe (*Weighted Methods per Class* - WMC)

Métrica que mede a complexidade de uma classe individual. Ao considerarmos os métodos de uma classe como sendo igualmente complexos, a métrica passa a ser o número de métodos definidos na classe.

Um método é contado apenas na classe que o definiu, não sendo contado nas classes que herdaram esse método. Desta forma, WMC pode ser definido como sendo o número de métodos públicos, privados e protegidos definidos na classe [41].

Assume-se que uma classe que tenha mais métodos é provavelmente mais complexa, tendo um maior impacto em suas subclasses e tem provavelmente maior aplicação específica, limitando a possibilidade de reuso.

Indica as classes que podem estar tentando fazer bastante trabalho, provendo muito mais funcionalidade e que consequentemente são mais difíceis de se testar, fazer a manutenção e mais propensas a falhas.

2.4.1.4 Resposta para uma classe (*Response for a Class* - RFC)

Métrica de acoplamento que consiste na soma do número de métodos da classe mais os métodos que são invocados diretamente por eles. É o número de métodos que podem ser potencialmente executados em resposta a uma mensagem recebida por um objeto de uma classe ou por algum método da classe [16]. A métrica avalia a complexidade da classe através do número de métodos e do volume de comunicação com outras classes [5].

A medida indica a complexidade da classe, uma vez que quanto mais métodos estiverem 'ligados' à classe (acessíveis dentro da hierarquia de classe), mais complexa é a mesma, o que consequentemente reflete nos esforços de testes requeridos e de manutenção da classe e, também, na tendência a falhas. Assim, assume-se que quanto maior o resultado de RFC, mais complexa é a classe.

Abaixo um exemplo para uma melhor compreensão da métrica:

```
public class meio_transporte {
```

```
      int quilometros_rodados;
      string cor;
      int ano;

      public string classificacao() {
        string classif;
        double media1= carro.calcula_media ();
        if (media1< 0) then classif = 'A';
        if (media1 = 0) then classif = 'B';
        if ((media1 > 0) && (media1 < 100)) then classif = 'C';
        if (media1> = 100) then classif = 'D';
        return classif;
      } // end classificacao
} // end meio_transporte

public class carro() extends meio_transporte {
        boolean acessorios; // se possui ou não acessórios adicionais
        boolean quatro_portas; // se tiver 4 portas ou não
        double media;
        double valor = 3000;

        public double calcula_media() {
            media  = (Quilometros_rodados/(2002 - ano));
            if (acessorios) then media = media +10;
            return media;
        } // end calcula_media

        public double calcula_valor() {
            if (ano > 1999) then valor = valor + 3000;
            if (quatro_portas) then valor = valor + 1000;
            return valor;
        } // end calcula_valor
} // end class carro
```

Tomando o exemplo acima, no qual é definida uma superclasse **meio_transporte** que tem o objetivo de classificar os meios de transportes nas categorias 'A', 'B', 'C' e 'D' de acordo com os valores obtidos da média entre 'quilômetros_rodados' e 'ano' de suas subclasses. O RFC para a superclasse **meio_transporte** é igual a 2, pois a mesma possui um método (**classificacao**) e através deste método, o método **calcula_media** é executado. O método **calcula_valor** não é contado pelo fato de não poder ser executado pela classe.

2.4.1.5 Acoplamento entre objetos (*Coupling Between Object* - CBO)

Métrica de acoplamento que mede o nível de acoplamento entre classes. Há acoplamento entre duas classes quando uma classe usa métodos e/ou variáveis de instância de outra classe. A medida é o número de classes às quais uma classe está acoplada de alguma forma, o que inclui o acoplamento baseado em herança.

Indica o potencial para reuso, uma vez que objetos pouco acoplados são 'mais independentes', contribuindo no projeto modular, sendo mais fácil de ser reusado em outras aplicações e promovendo o encapsulamento.

A medida também indica a complexidade da classe, o que consequentemente reflete nos esforços de testes requeridos e de manutenção e, também, na tendência a falha. Quanto maior o número de acoplamento, mais sensível a classe é para mudanças em outras partes do projeto, dificultando a manutenção, requerendo testes mais rigorosos e aumentando a probabilidade de se encontrarem falhas nela.

2.4.1.6 Falta de Coesão entre os métodos (*Lack of Cohesion in Methods* - LCOM)

Considerada uma métrica de coesão de uma classe, que é caracterizada pelo modo como os métodos da classe se relacionam com as variáveis de instância locais [17]. Mede o número de pares de métodos na classe que não compartilham variáveis de instância menos o número de pares de métodos que compartilham variáveis de instância. Quando o resultado é negativo, a métrica recebe o valor zero [16].

A falta de coesão nos métodos de uma classe é indesejável e implica que a classe poderia ser dividida em mais subclasses. Assim, o ideal é que LCOM tenha um valor baixo.

Quanto maior a coesão entre os métodos da classe, maior é o encapsulamento e menor a complexidade, sendo mais fácil manter a classe e tendo uma menor probabilidade de ocorrer falhas nela.

2.5 AVALIAÇÃO DAS MÉTRICAS OO

Após calcular as métricas obtemos dados quantitativos ou qualitativos de entidades do mundo real. Para utilizar estes dados é necessário analisar os resultados obtidos através de uma metodologia de avaliação a fim de associá-los a atributos de entidades do mundo real, de modo a descrevê-los segundo regras claramente definidas.

Como visto anteriormente, as métricas de software colaboram para melhor compreender a qualidade do produto. Assim, elas podem ser utilizadas no decorrer do desenvolvimento de um software para verificar uma propensão a falhas de determinadas partes do sistema, a fim de, se concentrar os testes nestas partes e corrigí-los, melhorando a qualidade do produto final. Em [46], algumas métricas OO foram utilizadas para este propósito.

As métricas de software também são úteis para planejar e avaliar o processo de software. Por exemplo, em [9] a técnica FP [4,30] foi utilizada para estimar o custo do projeto "Custos para Formação do Preço do Transporte". O projeto foi utilizado como um estudo de caso para simular o cálculo do custo de transporte de uma mercadoria, de uma cidade para outra. Os resultados mostraram que a partir de uma pontuação estabelecida, é possível estimar-se o tempo, custo e produtividade dos profissionais envolvidos com o projeto.

Nesta seção será descrita a metodologia de avaliação utilizada nos principais trabalhos da área de métricas OO para analisar-se os dados obtidos por uma variável de resposta binária. Neste trabalho, a variável de resposta binária indica se a métrica OO está relacionada à propensão de encontrar falhas em programas Java.

Uma alternativa para análise de variável resposta binária está baseada na construção de um modelo estatístico que serve para descrever o relacionamento entre uma variável resposta observada (variável dependente) e as variáveis explanatórias (variáveis independentes) [6].

O objetivo básico da modelagem é derivar uma representação matemática do relacionamento entre uma variável dependente e um número de variáveis independentes, com uma

medida de incerteza do relacionamento. O objetivo deve ser determinar se existe realmente um relacionamento entre uma resposta particular e um número de outras variáveis, ou estudar o padrão de algum relacionamento [6]. Os modelos estatísticos são essencialmente descritivos e, visto que são baseados em dados de experimento ou observação, podem ser descritos como modelos empíricos (diferente do modelo matemático) [6].

Na estatística existem dois métodos gerais mais importantes utilizados na estimação para ajuste dos modelos lineares: método de mínimos quadrados e método de máxima verosimilhança.

O método de estimação dos parâmetros mais utilizado é o da máxima verosimilhança, que consiste em determinar os valores dos parâmetros que maximizam a distribuição conjunta dos dados sob a suposição de independência entre os mesmos [6]. Este método é utilizado no modelo de regressão logística para estimar os valores dos c_{is} da Equação 2.1.

O **modelo de regressão logística** é o modelo logístico linear ajustado para explorar o relacionamento (se existir) entre uma variável resposta binária (variável dependente) e uma ou mais variáveis independentes [6]. Neste trabalho, a variável dependente é a ocorrência de falha na classe e as variáveis independentes são os valores das métricas na classe.

A técnica da regressão logística é bastante utilizada nos trabalhos de validação empírica das métricas orientadas a objetos, discutidas na Seção 2.5. Damos neste trabalho um breve resumo sobre esta técnica, para mais esclarecimentos o leitor pode consultar [6] e [18]. Outras técnicas de classificação, como Árvores de Classificação utilizada em [8], Redução do Conjunto Otimizado utilizada em [36] e Redes Neurais utilizada em [45], também poderiam ser utilizadas.

A próxima subseção descreve a técnica regressão logística e a Subseção 2.5.2 mostra alguns dos resultados da avaliação das métricas OO em alguns trabalhos.

2.5.1 Regressão Logística

Regressão logística é uma técnica de classificação padrão, baseada na estimação da máxima verosimilhança [18]. A seguinte equação demonstra um modelo de regressão

logística múltipla:

$$\pi(X_1, ..., X_n) = \frac{e^{(c_0+c_1X_1+...+c_nX_n)}}{1+e^{(c_0+c_1X_1+...+c_nX_n)}} \quad (2.1)$$

- π é a variável dependente. Neste trabalho, π é a probabilidade de uma falha estar relacionada com as métricas;
- os X_i são as variáveis independentes no modelo. Representam as métricas utilizadas; e
- os c_i são os coeficientes que deverão ser estimados usando-se o método de máxima verosimilhança. Para calculá-los existem atualmente vários pacotes, como: SAS, GLIM e Statistics. Estes pacotes utilizam-se de métodos numéricos, pois não há uma solução analítica (fórmula matemática) que calcule estes coeficientes.

O modelo de regressão logística simples é um caso especial, onde existe apenas uma variável independente.

A variável dependente π é uma probabilidade condicional e não é medida diretamente, como em outras técnicas de regressão, como a regressão linear. Um exemplo de probabilidade condicional é a probabilidade de se encontrar uma falha em uma classe como uma função das propriedades estruturais da classe, o que já foi feito em [14, 21, 36, 46].

O seguinte exemplo mostra este conceito. Um modelo de predição simples para a tendência a falhas de classes, como uma função de sua profundidade na árvore de herança (DIT). Se para DIT = 3, obtém-se $\pi(3) = 0.4$, interpreta-se que há uma probabilidade de 40% de se verificar uma falha em uma classe com DIT = 3 ou então, 40% das classes que possuem DIT = 3 contém uma falha.

Nos trabalhos, por exemplo [33, 46], em que são estudados o relacionamento entre as métricas e tendência a falha, tem sido comum encontrar a utilização de regressão logística simples para avaliação do relacionamento de cada uma das métricas isoladamente com relação à falha e depois a execução da regressão logística múltipla para avaliação da capacidade preditiva das métricas que têm sido apontadas como significantes na análise simples.

Exemplo de construção de um modelo estatístico, utilizando-se o pacote GLIM:

Suponha que se deseja verificar se existe algum relacionamento entre a probabilidade de um prendedor de uma aeronave falhar e a carga aplicada na aeronave. Neste caso, como existe apenas uma variável explanatória (carga aplicada), a regressão logística simples é utilizada.

Tabela 2.4: VALORES FORNECIDOS PELO PACOTE GLIM

Parâmetro	**Erro Padrão**	**Estimativa**
1	0.5457	-5.340
carga	0.0001575	0.001548

Os valores da estimativa e do erro padrão são fornecidos pelo pacote e o parâmetro é a variável independente. Para uma carga de, por exemplo, 2500 Kg, a estimativa de probabilidade de falha é de:

$$\pi = \frac{\exp(-5.340 + 0.00155 * 2500)}{1 + \exp(-5.340 + 0.00155 * 2500)}$$

$$\pi = 0.1877$$

O erro padrão (*standard error*) observado na Tabela 2.4 refere-se ao erro padrão relativo à estimativa da probabilidade de sucesso. A estimativa da probabilidade de sucesso que tem um grande erro padrão é menos precisa que a estimativa com menor erro padrão. Da mesma forma, o erro padrão é fornecido pelos pacotes.

Pode-se gerar os seguintes dados estatísticos, para cada uma das métricas aplicadas em cada uma das classes do sistema:

- **coeficiente (c_i):** coeficiente de regressão avaliado. Quanto maior o coeficiente em valor absoluto, mais forte o impacto (positivo ou negativo, de acordo com o sinal do coeficiente) da variável explicativa na probabilidade π de uma classe ser propensa a falha;

- $\Delta\Psi$: baseada na noção de razão de chance (*odds ratio*) [18], provê uma avaliação do impacto da métrica na variável resposta.

Mais especificamente, a razão da chance $\Psi(X)$ apresenta a razão entre a probabilidade de se ter uma falha e a probabilidade de não se ter uma falha quando o valor da métrica é X [46]. Como exemplo, se para um dado valor X, $\Psi(X)$ é 2, então é duas vezes mais provável que uma classe tenha uma falha do que ela não ter [46]. Quando dois conjuntos de dados binários são comparados, uma medida relativa da probabilidade de um sucesso em um conjunto com relação ao outro conjunto é a razão de chance [11]. Suponhamos que p_1 e p_2 são as probabilidades de sucesso em dois conjuntos, então a probabilidade de um sucesso no i-ésimo conjunto é $p_i/(1-p_i)$, i=1, 2. A razão de chance de um sucesso em um conjunto de dados binários com relação ao outro conjunto é denotado por [6]:

$$\Psi = \frac{p_1/(1-p_1)}{p_2/(1-p_2)}$$

Suponhamos os seguintes conjuntos de dados binários:

Tabela 2.5: RAZÃO DE CHANCE DE UM SUCESSO ENTRE DOIS CONJUNTOS DE DADOS BINÁRIOS

	Número de sucessos	Número de falhas
Conjunto de dados 1	a	b
Conjunto de dados 2	c	d

Tem-se:

$$\Psi = \frac{ad}{bc}$$

Por exemplo, pesquisadores gostariam de estimar a chance de um tumor ocorrer em ratos expostos ao cigarro comparado a ratos controlados (não expostos) [6]. Os dados deste exemplo podem ser vistos na Tabela 2.6.

- a chance de um tumor ocorrer em ratos expostos ao cigarro é: 21/(23-21)=10.50;
- para o grupo de ratos controlados é de 19/(32-19)=1.46;

Tabela 2.6: EXEMPLO REFERENTE À RAZÃO DE CHANCE

	Ratos com tumor	Ratos sem tumor
Ratos expostos	21	2
Ratos controlados	19	13

- a razão estimada de chance de ocorrer um tumor no grupo exposto relativo ao grupo controlado é dado por $\Psi = (21 * 13)/(2 * 19) \simeq 7.2$;
- interpretação: a chance de ocorrer um tumor no grupo exposto é aproximadamente 7 vezes maior do que no grupo controlado.

O valor de $\Delta\Psi$ é calculado pela Equação 2.2 [46]:

$$\Delta\Psi = \frac{\Psi(X+1)}{\Psi(X)} \tag{2.2}$$

Assim, $\Delta\Psi$ apresenta a redução/aumento na razão de chance quando o valor X aumenta em uma unidade, provendo uma noção do impacto da variável independente na resposta [46].

Tem-se ainda a medida de **Significância estatística (*p-value*)**. Ela é a medida básica da estatística que provê uma percepção precisa dos coeficientes estimados. Uma significância mínima de $\alpha = 0.05$ (probabilidade de 5%) tem sido utilizada para determinar se uma variável independente foi um preditor significante. De qualquer forma, a escolha de um nível particular de significância está ligada a uma decisão subjetiva e outros níveis tais como, $\alpha = 0.01$ ou $\alpha = 0.1$ são comuns. Além disso, quanto menor o nível de significância, menor o desvio padrão dos coeficientes estimados e maior é a confiabilidade do impacto das variáveis independentes calculadas. Em [18], os autores recomendam a utilização de um nível maior, como $\alpha = 0.25$, na análise simples, para não se cometer o erro de excluir uma variável independente que seria importante para o modelo múltiplo, descartando-a na análise simples, como feito em [33].

Com os resultados obtidos das métricas em cada uma das classes, pode-se avaliar quais métricas são úteis para a propensão a falhas em uma classe (análise de regressão simples)

e com qual grau de certeza elas são capazes de predizer a ocorrência ou não de falha (análise de regressão múltipla).

Para uma melhor compreensão da utilização das análises de regressão simples e múltipla, alguns dos dados coletados mais significativos em [33] serão utilizados para observação.

2.5.1.1 Análise Simples

A Tabela 2.7 mostra-nos os resultados da análise simples efetuada em [33].

Tabela 2.7: DADOS RESULTANTES DE [33] PARA ANÁLISE SIMPLES

Métrica	Coef. (c_i)	erro pad.	p	Ψ
RFC	0.102	0.018	<.0001	8.168
NIH-ICP	0.149	0.031	<.0001	9.272
OMMIC	0.191	0.047	<.0001	4.937
CBO	0.325	0.080	<.0001	2.012
LCOM	0.0025	0.0020	0.2135	1
LCOM_3	0.1474	0.0615	0.0164	2.054
Coh	-2.2314	0.7855	0.0045	0.654
LCC	-0.2743	0.3947	0.4870	0.902
DIT	0.6993	0.1614	0.0001	2.311
NOA	0.6811	0.1540	0.0001	2.307
NMO	0.5144	0.2296	0.0243	1.948
NMA	0.0681	0.0221	0.0021	1.710
Stmts	0.0189313	0.0030775	<0.0001	4.952
NM	0.0756322	0.0205779	<0.0001	2.026

- as métricas RFC, NIH-ICP, OMMIC e CBO são de acoplamento. Observa-se que as métricas RFC, NIH-ICP e OMMIC, que contam invocações a métodos, possuem valores altos para Ψ. Assim, conclui-se que a invocação a métodos é o principal mecanismo de acoplamento que tem um impacto na propensão de encontrar-se falha neste sistema estudado (outras métricas de acoplamento foram medidas).

- as métricas LCOM, LCOM_3, Coh e LCC são de coesão. As duas primeiras são medidas de coesão inversa (medem falta de coesão) e os valores dos seus coeficientes foram positivos. As duas outras são medidas de coesão corretas (medem a presença de coesão) e mostraram coeficientes negativos. Em cada caso, isto indica um aumento na probabilidade de propensão a falhas a medida que a coesão da classe diminui.

- as medidas LCOM_3 e Coh são significantes por possuírem p-value menores que 0.05 e validam a hipótese de que classes com alta coesão são menos propensas a conterem falhas.

- com relação às medidas de herança, todas as métricas foram consideradas significantes com p-value menor que 0.05, tendo um forte impacto na propensão de encontrar falhas. Através das métricas com coeficientes positivos conclui-se que: quanto mais profunda uma classe se encontra na hierarquia de herança, mais propensa é de conter falhas (DIT), quanto mais pais uma classe possui (mais métodos herda), mais propensa é de conter falhas (NOA), quanto mais métodos sobrescritos a classe possui, mais propensa é de conter falhas (NMO) e quanto mais métodos adicionados a classe possui, mais propensa é de conter falhas (NMA). Inversamente, a métrica NOC apresentou coeficiente negativo, mostrando que quanto mais filhos uma classe possui, menos propensa é de conter falhas. Porém, a métrica NOC não foi encontrada tendo um forte impacto, pelo fato do valor relativamente baixo de *p-value* (0.322).

- todas as medidas de tamanho (inclusive Stmts e NM) foram consideradas importantes com coeficientes positivos, validando a hipótese de que quanto maior o número de classes, maior a probabilidade de se encontrar falhas. A métrica que possui maior impacto na probabilidade de se encontrar falhas é Stmts ($\Psi = 4.952$).

2.5.1.2 Análise Múltipla

Nesta parte da avaliação, os autores, além de, buscarem obter o grau de certeza com que as métricas significantes, resultantes da análise simples, são capazes de predizer a ocorrência de falha, também procuraram saber se as métricas de acoplamento, coesão e herança eram complementares às métricas de tamanho, através da comparação de 3 modelos construídos. Um modelo apenas de métricas de tamanho (Modelo 1) , outro com métricas de acoplamento, coesão e herança (Modelo 2) e um outro com todas as métricas juntas (Modelo 3).

- para o Modelo 1, foram utilizadas 4 das métricas de tamanho e a verosimilhança foi de 127.94;

- O Modelo 1 foi aplicado para 113 classes de um sistema para comparar a predição da ocorrência de falha das classes com a real ocorrência de falha das classes. Uma classe foi classificada como "propensa à ocorrência de falha", se sua probabilidade de predição de conter uma falha é maior que 0.75 (percentual mínimo selecionado para equilibrar aproximadamente o número de classes propensas à ocorrência de falha da real ocorrência de falha nas classes). Na qualidade do ajuste (*goodness of fit*) foram identificadas 53 classes como propensa à ocorrência de falha, 32 das quais realmente apresentaram a ocorrência de falha (60% de exatidão);

- Para o Modelo 2 foram utilizadas 4 métricas de acoplamento e 3 métricas de herança (métricas de coesão não foram usadas por não terem se mostrado indicadores significantes na análise simples) e a verosimilhança foi de -70.62;

- o percentual mínimo selecionado para o Modelo 2 foi de 0.65 e o modelo se apresentou muito melhor na qualidade de ajuste: das 53 classes propensas à ocorrência de falha, 43 realmente tinham falhas (81% de exatidão). A alta qualidade do ajuste indica que as métricas de herança e acoplamento capturam dimensões estruturais com um forte relacionamento à ocorrência de falha, as quais vão além do tamanho das classes. Um modelo baseado nas propriedades de classes estrutural pode apresentar melhor qualidade que modelos baseados nas métricas de tamanho da classe somente, o que justifica um esforço extra para se coletar estas métricas;

- para o Modelo 3 foram utilizadas 4 métricas de acoplamento, 3 métricas de herança e 2 métricas de tamanho e a verosimilhança foi de -65.11;

- o percentual mínimo selecionado para o Modelo 3 foi de 0.7 e o modelo apresentou os seguintes resultados na precisão de ajuste: das 53 classes previstas à ocorrência de falha, 42 realmente apresentaram falhas (79% de exatidão);

- assim, o Modelo 2 foi considerado o melhor na previsão à ocorrência de falha, pela boa precisão apresentada (81% de exatidão).

2.5.2 Resultado da avaliação de métricas OO em alguns trabalhos

Em [46], os autores investigaram a propensão de uma classe conter falhas a partir dos dados coletados pelas seis métricas de CK. A Ferramenta GEN++ foi usada para extrair as métricas de CK diretamente do código fonte entregue no final da fase de implementação. Para validar estas métricas foram coletados dados de desenvolvimento de oito projetos em C++ de médio porte, construídos por estudantes com pouca experiência em OO. Os autores fizeram uma análise quantitativa e empírica dos dados coletados. Os resultados mostraram que as métricas NOC, RFC e DIT estão significativamente relacionadas com a propensão de encontrar falha. A métrica CBO está apenas medianamente relacionada com tendência à falha e a métrica WMC está pouco relacionada com a propensão de encontrar falha. Em todos os casos a métrica LCOM se mostrou insignificante para indicar uma classe como sendo propensa a conter falha. Além disso, eles afirmam que as métricas são melhores indicadores que as métricas tradicionais, uma vez que, podem ser aplicadas nas fases iniciais do ciclo de vida do software e que as métricas são indicadores complementares uma da outra, sendo relativamente independentes.

Em [33] foram explorados empiricamente os relacionamentos existentes entre medidas OO de acoplamento, coesão e herança e a probabilidade de propensão a falha nas classes do sistema durante a atividade de teste. Foram utilizados sistemas desenvolvidos por estudantes de graduação e pós-graduação sem experiência prévia ou treinamento do domínio da aplicação e calculadas várias métricas de acoplamento, coesão, herança e tamanho. Os sistemas foram desenvolvidos em C++, utilizando-se a metodologia OMT (fases de análise e projeto) e o ambiente de desenvolvimento GNU e OSF/MOTIF (fase de implementação). A análise dos dados coletados para cada medida foi descrito em quatro estágios: análises de distribuição de dados e estatísticas descritivas, análise de componente principal, construção do modelo de predição (regressão logística simples) e correlação com

o tamanho. Foram construídos modelos de predição múltiplos e, segundo os autores, os principais fatores que levam à propensão a falha são: tamanho da classe, a freqüência de invocações de métodos e a profundidade na hierarquia de herança. Como resultados do trabalho, os autores afirmam que:

- Muitas das medidas de acoplamento, coesão e herança capturam dimensões parecidas dos dados e isso ocorre pelo fato de que muitas medidas propostas na literatura são baseadas em idéias e hipóteses comparáveis, sendo assim, redundantes;
- Pela análise simples, muitas das medidas de herança e acoplamento são fortemente relacionadas à probabilidade de propensão a falha em uma classe, em particular, acoplamento através de invocações de método (CBO) e a profundidade de uma classe na hierarquia de herança (DIT). Medidas de coesão não mostraram ser indicadores significantes pela dificuldade de se medir o conceito e do fraco entendimento a que este atributo se propõe capturar;
- Pela análise múltipla, usando-se algumas das medidas de herança e acoplamento, modelos corretos podem ser derivados para predizer quais classes possuem a maior parte das falhas.

Em [16], a investigação empírica foi sobre um sistema de um produto de telecomunicações OO com aproximadamente 133.000 linhas de C++ desenvolvido usando-se o método de Shlaer e Mellor. Foram utilizadas as métricas CK para: considerar a sua efetividade e usabilidade em termos de facilidade para coleta de dados e utilidade para predizer falhas, estudar empiricamente um sistema OO não trivial, derivar sistemas de predição alternativos para predizer tamanho e erros usando medidas facilmente extraídas de documentação de projeto disponível e identificar fatores associados a altos níveis de falhas. Foram coletadas: variáveis na análise e no projeto que caracterizam a estrutura do sistema, variáveis gerenciais e dados de estatísticas descritivas. Sistemas de predição foram construídos e os autores obtiveram os seguintes resultados:

- os construtores são pouco utilizados nos sistemas OO. Há pouco uso de herança e

polimorfismo. Porém, as classes participantes da hierarquia de herança continham três vezes mais falhas do que as outras classes;

- classes que foram mais modificadas foram as que estavam em níveis mais distantes da raíz da árvore de hierarquia de herança;
- a métrica DIT pode ser usada para verificar classes que provavelmente apresentarão um grande número de falhas, uma vez que as classes que usam herança devem ser testadas completamente (provavelmente contêm falhas);
- as métricas CK não são facilmente coletadas na fase do projeto. Os autores apenas conseguiram coletar as métricas DIT e NOC, o que mostra uma desvantagem das métricas CK, uma vez que a coleta das mesmas dependem dos métodos de desenvolvimento e Ferramentas CASE utilizados.

Em [17] é investigado o relacionamento entre métricas OO e os esforços de manutenção que são medidos pelo número de linhas modificadas por classe. Foi feita uma análise estatística dos dados coletados através de 5 métricas CK (menos CBO), 3 métricas de acoplamento e 2 de tamanho que foram definidas. Foram utilizados dois software comerciais, desenvolvidos com uma linguagem de programação e projeto OO comercial que combina os construtores Ada com os construtores OO. Os dois softwares comerciais validaram o modelo de predição construído que mede os esforços de manutenção usando as métricas OO, coletadas do código fonte e do projeto. Para a análise dos dados coletados, alguns procedimentos de análise estatística foram usados, sendo o principal, a regressão linear múltipla. Os resultados demonstraram um forte relacionamento entre métricas e esforços de manutenção em sistemas OO e que os esforços de manutenção podem ser previstos pela combinação das métricas coletadas do código fonte.

Em [44], foram utilizadas as métricas CK, 1 métrica de tamanho e 1 métrica de produtividade em três aplicações comerciais de serviços financeiros OO para uso no gerenciamento de projetos, em questões como produtividade e custos. Análises exploratórias de dados empíricos relacionaram as métricas com produtividade, esforço de trabalho a ser feito novamente e esforço de projeto. As métricas foram calculadas de documentos

de projeto, planos de projeto e código fonte disponível para inspeção. Para a análise dos dados, 3 modelos de regressão foram construídos e analisados e a técnica estatísticas descritivas foi utilizada. Os resultados demonstraram:

- dados de métricas úteis podem ser coletados de sistemas escritos em uma variedade de linguagens de programação;
- as três aplicações tiveram um uso insignificante de herança (métricas DIT e NOC com valores mínimos), o que pode indicar que preferências de projeto apropriadas foram seguidas (pouco uso de herança);
- um alto relacionamento entre as métricas WMC, CBO e RFC, que pode ser explicado pelo fato de apenas a métrica CBO ter sido considerada como um preditor estatisticamente significante nestes modelos. Uma vez utilizando-se a métrica CBO, há pouco 'espaço' para WMC e RFC;
- valores altos para as métricas CBO e LCOM, o que foi associado com baixa produtividade, maior trabalho a ser feito novamente e maior esforço de projeto;
- as métricas CK através da validação empírica, mostraram oferecer um impacto significativo e quantitativo nas decisões de projeto OO no gerenciamento de produtividade e custo.

Outros trabalhos foram feitos investigando-se o relacionamento entre as métricas CK e propensão de encontrar falha. Os resultados variam para algumas métricas, mas para as métricas RFC e WMC a associação com a propensão de encontrar falha é encontrada em todos os trabalhos da área. Em [35], apenas a métrica NOC não foi associada com a propensão de ocorrência de falha (a métrica WMC não foi utilizada). Em [19], as métricas NOC e CBO não foram associadas com a propensão de ocorrência de falha e em [37] as métricas CBO e DIT não foram associadas à propensão de ocorrência de falha.

2.6 EXEMPLO DE APLICAÇÃO DAS MÉTRICAS

Para uma melhor compreensão da aplicação das métricas, algumas das métricas que serão utilizadas neste trabalho serão aplicadas em um simples exemplo no qual 3 classes (uma subclasse da outra) constroem uma caixa.

Código fonte do exemplo:

```
Class Caixa {
    public double largura;
    public double altura;
    public double profundidade;

    Caixa()
    {
        System.out.println (''Construindo caixa não inicializada !'');
        largura = -1;
        altura = -1;
        profundidade = -1;
    } // construtor

    Caixa (double larg, double alt, double prof)
    {
        System.out.println (''Construindo caixa inicializada !'');
        largura = larg;
        altura = alt;
        profundidade = prof;
    } // end construtor

    public double volume ()
    {
        return largura * altura * profundidade;
    }
} //end class Caixa

class CaixaComPeso extends Caixa {
    public double peso;

    CaixaComPeso ()
    {
        super (); // chama o construtor de Caixa
        peso = -1;
    } // end construtor

    CaixaComPeso (double pes, double larg, double alt, double prof)
    {
        super (larg, alt, prof); //chama o construtor de Caixa
        peso = pes;
        if (peso > 2) System.out.println (''Peso não permitido !'');
    } // end construtor

    CaixaVolume (double larg, double alt, double prof)
    {
        public double vol = Caixa.volume ();
```

```
        System.out.println (''O volume desta caixa é:'', vol);
    } // end CaixaVolume
} // end class CaixaComPeso

class CaixaEncomenda extends CaixaComPeso {
    public double custo;

    CaixaEncomenda ()
    {
        super (); // chama o construtor de Caixa
        custo = -1;
    } // end construtor

    CaixaEncomenda (double pes, double larg, double alt, double prof, double cust)
    {
        super (pes, larg, alt, prof); // chama o construtor de Caixa
        custo = cust;
        if (custo > 70) System.out.println (''Você pode escolher um '' +
                                            ''de nossos brindes !'');
    } // end construtor
} // end class CaixaEncomenda
```

A Figura 2.2 mostra a árvore de hierarquia do exemplo.

Figura 2.2: ÁRVORE DE HIERARQUIA

A Tabela 2.8 contém os valores de algumas das métricas, calculadas neste exemplo.

Considerações:

- A classe **Caixa** possui 3 métodos: 2 construtores (no caso de serem passados três parâmetros ou nenhum parâmetro) e um método público **volume ()**, para o cálculo do seu volume. Assim, o valor de WMC é 3 e de NPIM é 1;

Tabela 2.8: VALORES DAS MÉTRICAS DAS CLASSES DO EXEMPLO DE PROGRAMA

Métrica	Caixa	CaixaComPeso	CaixaEncomenda
WMC	3	3	2
DIT	1	2	3
NOC	1	1	0
CBO	0	1	2
RFC	3	6	4
LCOM	0	0	0
WMC_LOCM	12	10	7
NPIM	1	0	0
NIV	3	2	1
AMZ	4	3.3	3.5

- Como mostra a Figura 2.2, **Caixa** que é a raiz da árvore de herança, tem DIT = 1, enquanto que **CaixaComPeso** está no nível 2 e **CaixaEncomenda** no nível 3;
- **Caixa** possui uma subclasse imediata (filho), assim possui NOC = 1 e não faz chamada a nenhum outro método ou variável de outra classe, tendo CBO = 0;
- **CaixaEncomenda** está acoplada à classes **Caixa** e **CaixaComPeso** (CBO = 2) e a classe **CaixaComPeso** está acoplada apenas à classe **Caixa**;
- A classe **Caixa** têm apenas 3 métodos e destes não se pode fazer chamada a nenhum outro. Assim, RFC = 3;
- A classe **CaixaComPeso** possui apenas 3 métodos e através destes, mais 3 métodos podem ser executados (os 3 métodos da classe Caixa). Assim, RFC = 6;
- Já a classe **CaixaEncomenda**, possui 2 métodos e tem a possibilidade de através destes executar mais 2 métodos (os 2 métodos construtores de **CaixaComPeso**). Assim, CBO = 4;
- O valor de LCOM para a classe **Caixa** seria de -3. Os três pares de métodos (Caixa () & Caixa (double, double, double), Caixa () & Volume () e Caixa (double, double, double) & Volume ()) compartilham variáveis de instância (largura, altura e profundidade). Assim, o valor assumido é de 0;

- **Caixa** define 3 variáveis (NIV = 3). A classe **CaixaComPeso** define duas variáveis (peso e vol) tendo NIV = 2 e **CaixaEncomenda** define apenas uma variável (custo) possuindo, desta forma, NIV = 1;

- Para o cálculo da métrica LOCM não foram consideradas as linhas em branco, de comentários e de início/fim de bloco. Desta forma, o valor LOCM para a classe **Caixa** é de 12, da classe **CaixaComPeso** é de 10 e da classe **CaixaEncomenda** é de 7;

- Através da divisão da métrica LOCM por WMC obtém-se o valor da métrica AMZ. Sendo assim, o valor AMZ para a classe **Caixa** é igual a 4, para a classe **CaixaComPeso** é igual a 3.3 e para a classe **CaixaEncomenda** é igual a 3.5.

CAPÍTULO 3

ANÁLISE ESTÁTICA DE BYTECODE JAVA

Como destacado anteriormente, as métricas OO são coletadas a partir do projeto de classes ou do código fonte das mesmas. Entretanto, nas situações em que não se dispõe do código fonte e, menos ainda, do projeto das classes, fica impossibilitada a coleta das métricas OO para a obtenção de uma melhor qualidade do software e o planejamento do processo de desenvolvimento do mesmo. Para estas situações, os programas objeto Java podem ser utilizados para a obtenção das informações relativas às características de orientação a objetos que são utilizadas nas métricas OO e que, normalmente, são obtidas no código fonte.

Neste capítulo, serão descritos algumas das características da Ferramenta JaBUTi [48], uma ferramenta de tese baseada no fluxo de controle e de dados para programas Java que realiza uma análise estática em nível de bytecode Java e foi aproveitada neste trabalho para a implementação de algumas das métricas de software e como estudo de caso (Seção 5.2).

Na próxima seção será apresentado um breve resumo referente ao bytecode Java útil na compreensão deste trabalho. O leitor pode consultar o livro de Lindholm e Yellin [13] para uma referência completa sobre a especificação JVM, como também sobre o conjunto de instrução bytecode. A Seção 3.2 apresenta algumas das características e funcionalidades da Ferramenta JaBUTiMais informações sobre as outras funcionalidades providas pelo JaBUTi podem ser encontradas em [48].

3.1 BYTECODE JAVA

Uma das principais razões pelas quais a linguagem Java tem se tornado popular é a independência de plataforma provida pelo ambiente tempo de execução, a *Java Virtual Machine* (JVM). Compilando um programa Java, um conjunto de arquivos objetos é obtido, um para cada classe. O então chamado "arquivo classe" consiste em uma rep-

resentação binária portável que contém dados de classe relacionados, tais como:

- O nome da sua super-classe;
- O tipo e a assinatura de seus métodos;
- O tipo e o nome das suas variáveis de instância e de classe;
- Os métodos de outras classes que são invocados;
- A relação de classes da qual esta classe depende.

A JVM tem uma estrutura orientada a pilha. Esta cria uma pilha de operando local para invocar todos os métodos que efetuam a execução de instruções bytecode.

Instruções bytecode podem ser vistas como uma linguagem parecida com *assembly* que retém informação de alto nível do programa. O conjunto completo de instruções bytecode para a especificação JVM pode ser obtido em [13]. Uma instrução bytecode é apresentada por um one-byte opcode, seguido por valores operandos caso existam. Cada one-byte opcode possui um mnemônico mais significativo que ele próprio. Cada instrução lida com um tipo de dado representado explicitamente por uma letra que precede o nome mnemônico. Por convenção utiliza-se a letra i para integer, l para long, s para short, b para byte, c para char, f para float, d para double e a para reference. Como exemplo, a instrução que coloca um valor inteiro one-byte no topo da pilha operando JVM tem o opcode 16, o mnemônico bipush e requer um operando (o valor inteiro one-byte). A instrução bipush 9 é responsável para colocar o valor 9 no topo da pilha operando JVM.

Antes de executar um determinado programa Java, a JVM carrega e cria uma classe inicial, que é especificada de forma dependente de implementação pelo uso do *bootstrap* da classe **loader**. Após a classe inicial ter sido inicializada, o método classe público **void main(String[])** é invocado. A execução deste método resulta no carregamento e execução das classes adicionais e interfaces e na invocação de métodos adicionais [13].

Durante a execução, a JVM cria um frame de pilha local para cada invocação de método. Basicamente, ignorando as exceções, a interpretação do bytecode é feita pela 1)

dedução de um *opcode*, 2) dedução de cada um de seus operandos (caso existam) e 3) execução da ação para o bytecode [13].

3.2 JaBUTi

A Ferramenta JaBUTi [47] (*Java Bytecode Understanding and Testing*) foi projetada com o objetivo principal de permitir a aplicação direta de critérios de teste estrutural em programas objeto Java. A principal razão para se trabalhar em um nível mais baixo deve-se ao fato de, no caso da indisponibilidade do código fonte, seja possível derivar e utilizar requisitos de teste estrutural para garantir a qualidade de um dado conjunto de teste [48].

A finalidade da JaBUTi é prover tanta informação quanto for possível para que o usuário obtenha um melhor entendimento do programa, da mesma forma se o código fonte original estiver disponível. Nesta seção explicaremos, de forma suscinta, algumas das características da Ferramenta JaBUTi e como ela faz a análise estática no nível de bytecode Java através de um exemplo.

JaBUTi é uma ferramenta de teste baseada no fluxo de controle e fluxo de dados para programas Java e componentes Java. A ferramenta implementa dois critérios de teste baseados no fluxo de controle (todos os nós e todos os ramos) e um critério baseado no fluxo de dados (todos os usos) que podem ser usados para garantir a qualidade de um dado programa/componente Java [48].

O teste baseado no fluxo de controle utiliza uma representação de programa chamada de grafo de fluxo de controle (GFC) para selecionar os requisitos de teste [50]. Por exemplo, todos os nós e todos os ramos são conhecidos critérios de teste de fluxo de controle que requerem que todos os nós GFC (correspondentes a todos os comandos no programa) e todos os ramos GFC (correspondentes a todas as decisões no programa) sejam executados por, no mínimo, um caso de teste [40].

Critérios de fluxo de dados foram desenvolvidos como alternativas para o critério de fluxo de controle [12]. Em geral, os critérios de teste de fluxo de dados utilizam um grafo de definição e uso (*def-use*), que é um GFC estendido com informação sobre o conjunto

de variáveis definidas e usadas [12].

A Ferramenta JaBUTi faz a análise de arquivos de classe Java e deriva diversas informações, entre elas:

- A estrutura hierárquica do programa sob análise. A partir de uma determinada classe "raiz", a ferramenta descobre quais são as demais classes e interfaces requeridas e constrói a estrutura hierárquica. O testador pode escolher quais pacotes devem ser considerados dentro do escopo do programa, incluindo, por exemplo, classes localizadas em bibliotecas de terceiros. Como padrão, assume-se que as classes de sistema (da API Java) estão fora do escopo do programa. A Figura 3.1 mostra um exemplo;

- Para cada classe dentro do escopo do programa, pode-se obter quais são os métodos definidos; e

- Para cada método nas classes dentro do escopo do programa, é possível gerar-se o grafo de fluxo de controle e obter-se informação sobre definições e usos de variáveis dentro do método.

Com essas informações é possível, através da instrumentação direta do bytecode, executar casos de teste e colher informação sobre a cobertura destes em relação ao programa sob teste. É possível, também, obter-se informações sobre muitas das métricas OO propostas na literatura, conforme será descrito no Capítulo 4.

Um ponto importante a se notar é que a escolha da estrutura de programa comentada acima, na qual define-se um conjunto de classes que fazem parte do escopo do programa sob análise, ignorando-se as classes que não são de interesse, tem influência em algumas das métricas implementadas, a saber, naquelas métricas relacionadas com a hierarquia de classes. Por exemplo, na Figura 3.1, ambas as classes *Graph* e *GraphNode* têm valor 1 para a métrica DIT, independentemente do número de subclasses que possuam fora do escopo do programa.

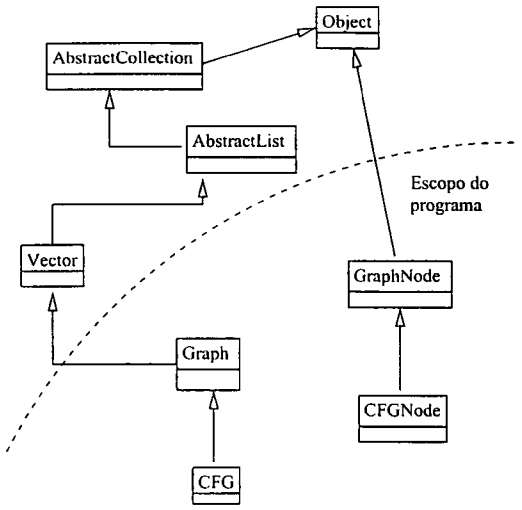


Figura 3.1: EXEMPLO DA ESTRUTURA DE UM PROGRAMA NA FERRAMENTA JaBUTi

3.2.1 Como executar a análise de cobertura

Nesta subseção, serão descritos de forma suscinta os passos necessários para a condução da atividade de teste utilizando-se a JaBUTi. Esta descrição está baseada em [49].

Através de um simples exemplo adaptado pelos autores, foram detalhados estes passos necessários. No exemplo, as classes **VendingMachine** e **Dispenser** implementam uma típica máquina de venda capaz de liberar itens específicos para um cliente sob certas condições. As operações que um cliente pode executar são: (1) inserir uma moeda de 25 centavos na máquina (VendingMachine.insertCoin()); (2) pedir à máquina para o retorno de moedas inseridas e não consumidas (VendingMachine.returnCoin()) e (3) pedir à máquina sobre a venda de um item específico (VendingMachine.vendItem()). Quando o item questionado não está disponível, o crédito é insuficiente ou a seleção é inválida, a máquina sinaliza um erro, indicando que a operação requisitada não pode ser executada.

Os autores recomendam primeiramente criar-se um projeto, que inclui um arquivo *class*, denominado base de classe, do qual a ferramenta extrai a hierarquia de classe completa, incluindo classes de sistema e as classes definidas pelo usuário, necessárias para executar esta base de classe. A Figura 3.2 mostra a caixa de diálogo "*JaBUTi Project Manager*", na qual a esquerda, estão as classes definidas pelo usuário, necessárias para executar a base de classe **VendingMachine**. Pode-se observar que as classes **Vending-**

Machine e **Dispenser** foram selecionadas para serem instrumentalizadas.

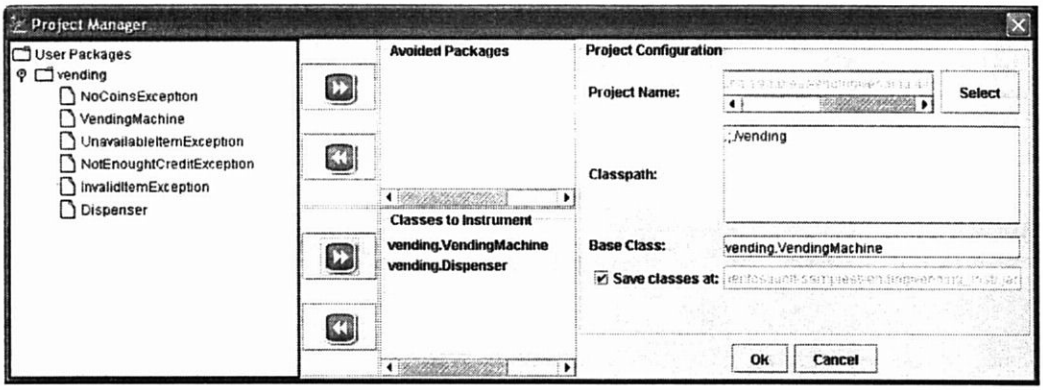


Figura 3.2: CAIXA DE DIÁLOGO "*JaBUTi PROJECT MANAGER*"

Após criar um projeto, segundo os autores, os principais passos necessários para executar-se a análise de cobertura das classes sobre o teste (CUTs) são [49]:

- gerar o grafo *def-use* (DUG) de cada método para cada CUT;
- derivar os requisitos de teste (TRs) de cada DUG, considerando diferentes critérios de teste;
- instrumentalizar os CUTs e executar as classes instrumentalizadas para coletar informação da execução traçada (ER);
- utilizar o ER para avaliar quais TRs foram cobertos; e
- gerar relatórios de cobertura para avaliar a qualidade do conjunto de teste.

Estes passos são descritos na Figura 3.3.

Após a criação de um projeto, JaBUTi contrói o DUG de cada método. Por exemplo, a Figura 3.4 mostra parte do DUG do método **Dispenser.dispense()**.

Como já comentado, a ferramenta não requer o código fonte para executar suas atividades, mas, quando o mesmo está disponível, JaBUTi permite mapear a informação obtida do bytecode para o código fonte, facilitando o entendimento de uma dada aplicação/componente. Por exemplo, a Figura 3.5 descreve o código fonte e parte do bytecode do método **Dispenser.dispense()**.

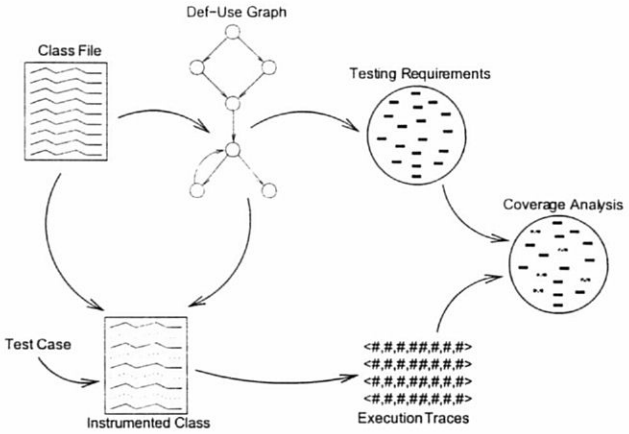


Figura 3.3: PRINCIPAIS FUNCIONALIDADES DO JaBUTi

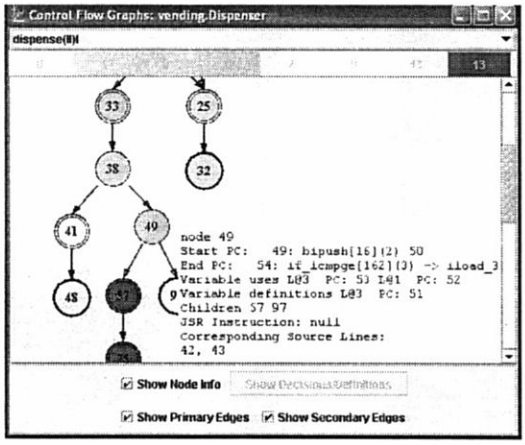


Figura 3.4: GRAFO *DEF-USE* DO MÉTODO **Dispenser.dispense**

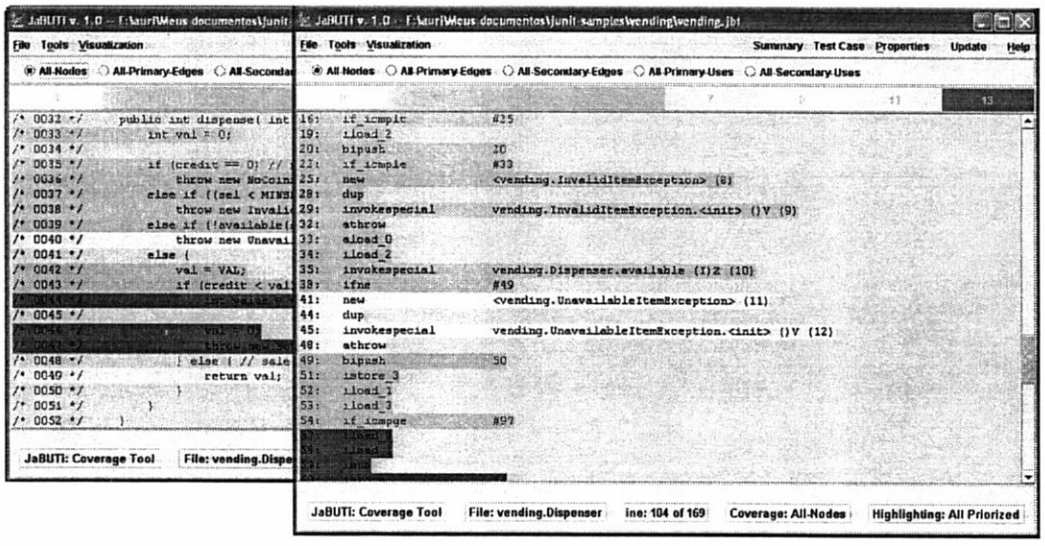


Figura 3.5: BYTECODE JAVA DO **Dispenser.dispense()**

Para derivar os TRs de cada DUG, os desenvolvedores implementaram quatro diferentes critérios de teste intra-métodos baseados no fluxo de controle (todos os nós primários, todos os nós secundários, todos os ramos primários e todos os ramos secundários) e dois diferentes critérios de teste intra-métodos baseados no fluxo de dados (todos os usos primários e todos os usos secundários), para gerar requisitos de teste que garantem a qualidade ou guiam na geração de um conjunto de teste. Os autores fizeram uma distinção entre os dois diferentes tipos de ramos com o intuito de representar o mecanismo de tratamento de exceção do Java.

Através da utilização do conceito de dominadores e super-blocos [2], a ferramenta designa diferentes pesos para os requisitos de cada critério de teste. Os pesos foram representados por diferentes cores e provêm sugestões para o testador sobre qual parte do bytecode, código fonte ou DUG deve ser coberto primeiramente, para a cobertura mais rápida de um dado critério de teste [49].

Para coletar-se a informação da execução dinâmica traçada, os desenvolvedores criaram uma extensão da classe **loader**, que executa a instrumentação dos CUTs, antes de carregá-los e executá-los. Instrumentando-se os CUTs, informação *trace* é gerada e armazenada em um arquivo *trace*. Cada execução da classe **loader** do JaBUTi representa um novo caso de teste e gera informação *trace* que representa a seqüência de nós DUG executados por cada caso de teste.

Através da Figura 3.6 os autores exemplificam este processo. A Figura 3.6(b) ilustra como invocar a extensão da classe **loader** para coletar-se a execução *trace* de um novo caso de teste. **ProaberLoader** é a classe **loader** do JaBUTi, o parâmetro **-P** especifica o nome do arquivo de projeto, **vending.jbt**, neste caso. **VendingMachine** é o nome da classe a ser executada e **input1** o parâmetro necessário para executar a aplicação **VendingMachine**. O caso de teste **input1** (Figura 3.6 (a)) corresponde ao caso de teste que executa o nó com maior peso mostrado na Figura 3.4, o nó 57.

Após terminada a execução da classe **loader** do JaBUTi um arquivo *trace* (**vending.trc**) foi gerado contendo a execução *trace* de **input1**. O arquivo *trace* é lido pelo JaBUTi para avaliar sua cobertura com respeito aos requisitos de teste de cada critério

$ cat input1 insertCoin vendItem 5	$ java probe.ProberLoader -P vending.jbt vending.VendingMachine input1 Project Name: vending.jbt Trace File Name: vending.trc VendingMachine ON Current value = 25 Current value not enough to buy item 2. VendingMachine OFF
(a)	(b)

Figura 3.6: CLASSE **loader** DO JaBUTi: (a) ENTRADA DO CASO DE TESTE E (b) SAÍDA DE **VendingMachine**.

de teste. Então, os pesos são recalculados e novos casos de teste podem ser gerados para aumentar a sua cobertura [49].

A Figura 3.7 mostra a versão alterada do bytecode previamente mostrado na Figura 3.5. Os autores observaram que as instruções bytecode dos *offsets* 57 a 60, os quais compõem o conjunto de instruções no nó 57, têm um peso zero, indicando que eles foram cobertos e que os requisitos com os maiores pesos neste momento, foram os nós 25 e 41, ambos com peso dois. Desta forma, eles desenvolveram dois casos de teste adicionais para cobrir tais requisitos, aumentando a cobertura do critério todos os nós.

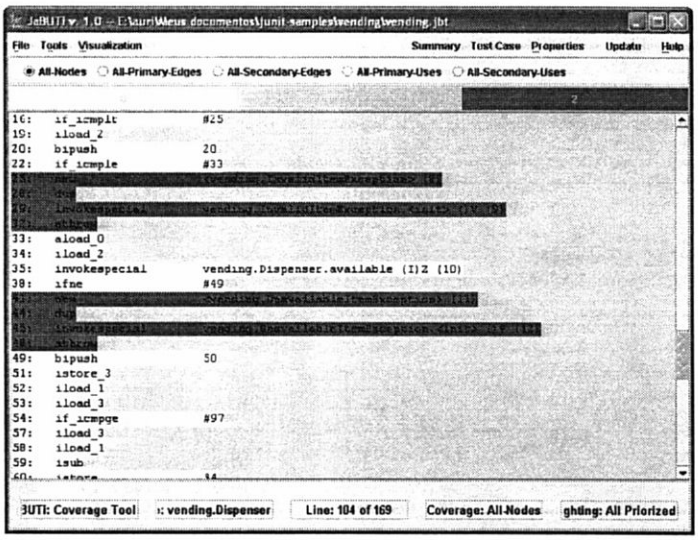
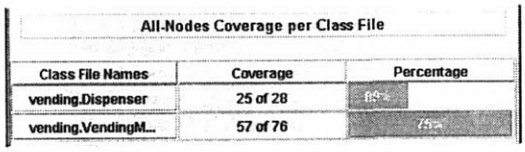


Figura 3.7: PESOS ALTERADOS PARA A CLASSE **Dispenser**

JaBUTi também provê diferentes relatórios de teste para ajudar o testador a decidir se é preciso gerar novos casos de testes para aumentar a cobertura, ou se a cobertura corrente já é o suficiente. A ferramenta permite a geração de relatórios resumidos para

cada critério, arquivo classe, método e caso de teste.

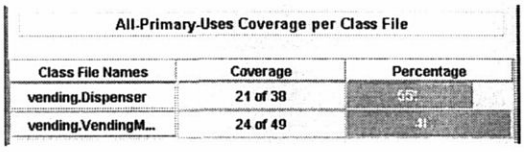
A Figura 3.8 ilustra o relatório resumido por arquivo, resultante dos 3 casos de teste desenvolvidos, considerando dois diferentes critérios de teste: todos os nós e todos os usos primários. Segundo os autores, em [49], os outros relatórios de cobertura são similares a estes. Eles observaram que para o critério todos os nós, 25 dos 28 nós da classe **Dispenser** foram cobertos, os quais representam 89% do número total de requisitos e para o critério todos os usos primários, 21 das 38 associações *def-use* (55%) foram cobertas.



All-Nodes Coverage per Class File

Class File Names	Coverage	Percentage
vending.Dispenser	25 of 28	89%
vending.VendingM...	57 of 76	75%

(a) POR ARQUIVO: TODOS OS NÓS



All-Primary-Uses Coverage per Class File

Class File Names	Coverage	Percentage
vending.Dispenser	21 of 38	55%
vending.VendingM...	24 of 49	49%

(b) POR ARQUIVO: TODOS OS USOS PRIMÁRIOS

Figura 3.8: RELATÓRIO RESUMIDO POR ARQUIVO

CAPÍTULO 4

MÉTRICAS IMPLEMENTADAS

Na Ferramenta JaBUTi, a classe que implementa as métricas faz parte do pacote **metrics** e denomina-se **Metrics**. A classe **Metrics** possui 38 métodos e implementa as 27 métricas calculadas na Ferramenta JaBUTi utilizando a estrutura do programa e as informações coletadas para cada classe e método. Cada classe é representada na estrutura por uma classe **RClassCode**, que terá o seu conjunto de métricas. Das métricas implementadas:

- 10 são métricas de LK;
- 4 são variações das métricas LK, que aproveitam o fato de serem aplicadas no bytecode e não no programa fonte;
- 6 são métricas de CK;
- 5 são variações das métricas CK; e
- 2 são variações da métrica tradicional Complexidade Ciclomática de McCabe (CC).

Nas próximas seções serão apresentadas uma descrição suscinta de cada uma das métricas e como as mesmas foram implementadas no JaBUTi pela classe **Metrics**.

4.1 MÉTRICAS DE LK

Número de métodos de instância públicas na classe - NPIM

A métrica *Number of public instance methods in a class* (NPIM) é calculada através da contagem do número de métodos de instância públicos na classe, incluindo os construtores.

A ferramenta calcula a métrica verificando para cada método da classe se o mesmo é público (através do método booleano **isPublic()**) e não é um método abstrato (método

booleano **isAbstract()**) e nem estático (método booleano **isStatic()**). O número total dos métodos públicos que não são *abstract* e nem *static* resulta no valor final da métrica para a classe.

Número de variáveis de instância na classe - NIV

A métrica *Number of instance variables in a class* (NIV) é calculada através da contagem do número de variáveis de instância na classe, o que inclui as variáveis *public*, *private* e *protected* disponíveis para as instâncias.

JaBUTi calcula a métrica somando todas as variáveis na classe que não são estáticas (método booleano **isStatic()**). Ocorre justamente o contrário da métrica NCV descrita mais abaixo. A soma das métricas NIV e NCV resulta no número total de variáveis na classe.

Número de métodos de classe na classe - NCM

A métrica *Number of class methods in a class* (NCM) é calculada através da contagem do número de métodos estáticos na classe.

O cálculo da métrica na ferramenta é feito através da verificação para cada método da classe se o mesmo é estático e não abstrato. A soma dos métodos que são estáticos e não abstratos resulta no valor final da métrica para a classe.

A métrica WMC_1 descrita abaixo difere desta por somar todos os métodos na classe, inclusive os métodos não estáticos.

Variação de NCM - denotada por NCM_2

Conta-se o número de métodos públicos estáticos na classe a fim de se obter o valor desta métrica. Observe que o valor desta métrica será sempre igual ou menor que o valor da métrica NCM.

JaBUTi calcula a métrica verificando para cada método da classe se o mesmo é estático e público, e não é um método abstrato. A soma dos métodos que são estáticos, públicos

e não abstratos resulta no valor final da métrica para a classe.

Número de variáveis de classe na classe - NCV

A métrica *Number of class variables in a class* (NCV) é calculada através da contagem do número de variáveis estáticas na classe.

JaBUTi calcula a métrica verificando se cada uma das variáveis na classe é estática e soma todas as que são.

Número médio de parâmetros por método - ANPM

A métrica *Average number of Parameters per method* (ANPM) é calculada através da razão entre a somatória do número de parâmetros de cada método da classe pelo número total de métodos da classe. Construtores são considerados.

A métrica é calculada no JaBUTi da seguinte forma:

- para cada método da classe, o método **getArgumentNames()** pega cada um dos nomes de parâmetros e coloca-os no array **dummy[]**;
- o tamanho do array **dummy[]** é somado por um contador inteiro denominado **contPar**;
- este processo é repetido até que todos os métodos da classe sejam verificados;
- se a classe não tem nenhum método, o valor da métrica é de -1;
- o valor final da métrica é dado pela razão do contador **contPar** pelo número de métodos da classe.

Variação de ANPM - denotada por MNPM

O valor da métrica equivale ao número máximo de parâmetros nos métodos da classe.

A métrica é calculada de forma parecida com a métrica ANPM descrita anteriormente. Porém, após pegar o tamanho do array **dummy[]** é feita uma comparação do tamanho

deste vetor com o valor inteiro que está na variável **max**. Se o último valor calculado for maior, a variável **max** recebe este valor. Ao final da verificação de todos os métodos da classe, o valor de **max** é o valor final da métrica.

Uso de herança múltipla - UMI

A métrica *Use of multiple inheritance* (UMI) não foi implementada na ferramenta pelo fato de a linguagem Java não permitir o uso de herança múltipla.

Foi implementada uma variação desta métrica denotada por NII, descrita logo abaixo.

Número de interfaces implementadas - NII

A métrica *Number of Interfaces Implemented* (NII) é calculada pela soma do número de interfaces implementadas pela classe. Métrica aplicada à classe, calculada no JaBUTi da seguinte forma:

- o método **getInterfaceNames()** pega cada um dos nomes de interfaces implementadas pela classe e coloca-os no array **dummy**[];
- se o array **dummy**[] estiver vazio então o valor da métrica será igual a 0;
- do contrário, o valor final da métrica é dado pelo tamanho do array **dummy**[].

Tamanho médio do método - AMZ_LOCM

A métrica *Average method size* (AMZ_LOCM) é calculada através da divisão entre a soma do número de linhas de código dos métodos da classe pelo número de métodos na classe (soma dos métodos de instância e classe). Métodos construtores, também são considerados. Note-se que esta métrica relaciona-se realmente ao número de linhas de código no programa fonte, uma vez que do bytecode é possível extrair este valor.

A métrica é calculada no JaBUTi da seguinte forma:

- para cada método da classe, executa-se o método **LOCM()** que utiliza o método **getLineNumbers()** para calcular o número de linhas de código do programa;

- o resultado de **LOCM** é somado por um contador inteiro denominado **theValue**;
- este processo é repetido até que todos os métodos da classe sejam verificados;
- se a classe não possui nenhum método, o valor da métrica é de -1;
- o valor final da métrica é dado pela razão do contador **theValue** pelo número de métodos da classe.

Variação de AMZ_LOCM - denotado por AMZ_SIZE

É semelhante a AMZ_LOCM, embora utilize o número de instruções do bytecode como tamanho do método e não o número de linhas de código do programa fonte.

JaBUTi calcula esta métrica da mesma forma que para a métrica AMZ_LOCM, com a diferença de utilizar o método **SIZE()** no lugar do método **LOCM()**. O método **SIZE()** utiliza os métodos **getInstructionList()** e **getLength()** para contar o número de instruções do bytecode do método.

Número de métodos sobrescritos na subclasse - NMOS

A métrica *Number of methods overridden by a subclass* (NMOS) é calculada através da contagem do número de métodos definidos na subclasse com a mesma assinatura de um método em uma das suas superclasses (diretas ou indiretas). Nesta e nas demais métricas que requerem um passeio nas superclasses de uma classe, somente as classes no escopo do programa são utilizadas na busca. Além disso, construtores (que no programa objeto recebe sempre o nome <*init*>) e inicializadores estáticos (<*cinit*>) não são contados.

A ferramenta utiliza o método **findMethInClass(methods[i], rcc.getSuperClass(), true)**. que requer como parâmetros, respectivamente, um dos métodos da subclasse, as informações da superclasse comparada e o método é verdadeiro se o nome do método coincidir com o nome de um método da superclasse comparada. Então, soma-se valor 1 ao contador **cont** e o próximo método da subclasse é comparado. Após fazer esta verificação para cada um dos métodos da subclasse o valor de **cont** é retornado. A comparação é

feita entre o nome de cada um dos métodos da subclasse com todos os nomes de cada uma de suas superclasses.

Número de métodos herdados pela subclasse - NMIS

A métrica *Number of Methods inherited by a subclass* (NMIS) é calculada através da contagem do número de métodos herdados pela subclasse de suas superclasses. Da mesma forma, que na métrica NMOS, os construtores e inicializadores estáticos não são considerados.

JaBUTi também utiliza o método **findMethInClass()**. Porém, compara-se cada um dos nomes de métodos de cada uma de suas superclasses com cada um dos nomes de método da subclasse. Se o nome não coincidir, então o contador **cont** é incrementado. O valor final da métrica é o valor de **cont**.

Número de métodos adicionados pela subclasse - NMAS

A métrica *Number of Methods added by a subclass* (NMAS) é calculada através da contagem do número de novos métodos adicionados pela subclasse, os construtores e inicializadores estáticos, também não são considerados.

A ferramenta calcula a métrica de forma idêntica como calcula a métrica NMOS, mas apenas incrementa o contador **cont** se o método da subclasse não coincide com nenhum dos nomes de métodos de cada uma de suas superclasses.

Índice de Especialização - SI

A métrica *Specialization index* (SI) é calculada através da divisão entre o resultado da multiplicação de NMOS e DIT (métrica de CK) pelo número total de métodos.

JaBUTi calcula a métrica simplesmente executando os métodos **NMOS()** e **DIT()** na classe e efetuando o cálculo através da multiplicação dos valores retornados por estes métodos dividido pelo número de métodos da classe.

4.2 MÉTRICAS DE CK

Número de Filhos - NOC

A métrica *Number of Children* (NOC) é calculada através da contagem do número de subclasses imediatas subordinadas à classe na árvore de hierarquia de herança.

JaBUTi utiliza o método **getSubClasses()** que retorna a distância da borda do programa ou o valor de -1 se a classe não estiver no programa.

Profundidade da Árvore de Herança - DIT

A métrica *Depth of Inheritance Tree* (DIT) é o maior caminho da classe à raiz na árvore de hierarquia de herança. Interfaces também podem ser consideradas, ou seja, o caminho através de uma hierarquia de interfaces também pode ser o que dá a profundidade de uma classe. Como a representação do programa utilizada não inclui todas as classes até a raiz da árvore de hierarquia, é utilizado o caminho da classe até a primeira classe que não pertence à estrutura do programa.

A ferramenta pega as interfaces através da execução do método **getInterfaces()** para verificar através do método **levelOf()**, o nível em que a classe se encontra na árvore de hierarquia de interfaces e compara com o nível em que a classe se encontra na árvore de hierarquia de herança (com o mesmo método **levelOf()**). O maior valor é retornado como o valor final da métrica.

Número ponderado de métodos por classe - WMC

A métrica *Weighted Methods per Class* (WMC) é calculada através da soma da complexidade de cada método. Na proposição original da métrica não se define qual tipo de complexidade pode ser utilizada. Desta forma, neste trabalho são definidas e implementadas quatro variações para a mesma, que estão descritas logo abaixo.

WMC_1

Utiliza o valor 1 como métrica de complexidade de cada método. Número de métodos na classe.

Através do método **getMethods()**, os métodos da classe são colocados em um array. Para cada método é verificado se o mesmo é abstrato, em caso positivo, não é somado na métrica.

WMC_CC

Utiliza a métrica de Complexidade Ciclomática de McCabe (CC) para a complexidade de cada método. Soma-se o valor CC de cada método da classe. Os construtores são considerados.

Para cada método, JaBUTi calcula o valor CC através do método **CC**. Se este valor for menor que zero, então o valor de CC é de -1, senão o valor de CC é somado ao contador **theValue**. O valor final do **theValue** é equivalente ao valor da métrica.

O método **CC()** constrói o GFC do método com o método **getGFC**. Através do método **SIZE()**, obtém-se o número de nós que o grafo possui e para cada nó **GFCNode** do grafo, o número de ramos é obtido através dos métodos **elementAt()** e **getPrimNext()**.

WMC_LOCM

Utiliza-se a métrica Linhas de Código (LOCM) para a complexidade de cada método. Soma-se o valor LOCM de cada método da classe.

JaBUTi usa o método **LOCM()** para o cálculo do número de linhas de código de cada método e soma este valor ao contador **theValue**. O valor final de **theValue** equivale ao valor da métrica.

WMC_SIZE

Utiliza-se o número de instruções para a complexidade de cada método.

O método **SIZE()** é usado pela ferramenta para o cálculo do número de instruções bytecode de cada método e soma este valor ao contador **theValue**. O valor final de **theValue** equivale ao valor da métrica.

Falta de Coesão entre os métodos - LCOM

A métrica *Lack of Cohesion in Methods* (LCOM) é calculada através da contagem do número de pares de métodos na classe que não compartilham variáveis de instância menos o número de pares de métodos que compartilham variáveis de instância. Quando o resultado é negativo, a métrica tem o valor zero. Os métodos estáticos não são considerados na contagem, uma vez que, somente as variáveis de instância são tomadas. Construtores são considerados. São aplicadas duas variações para a métrica, que serão identificadas logo abaixo.

JaBUTi utiliza o método **LCOM_0()**, que para cada um dos métodos, primeiramente exclui os métodos estáticos da análise através do método **isStatic()** e monta o GFC através da execução do método **getCFG()**. A coleção de definições e usos das variáveis de instância no método é retornada pelo método **findDefUse()**. Para se manter somente os acessos não estáticos, utiliza-se o método **startsWith("L@0.")**. Após verificar-se todas as *def-uses* de cada um dos métodos, compara-se entre pares de métodos, as variáveis de instância que não são compartilhadas (incrementa-se o contador **notShare**) e as que são (incrementa-se o contador **doShare**). Se a diminuição **notShare** de **doShare** resultar em um número negativo, o método retorna o valor 0.

LCOM_2

Considera somente a coesão entre métodos estáticos, levando-se em conta obviamente, o compartilhamento de variáveis estáticas.

JaBUTi também utiliza o método **LCOM_0()** para cada um dos métodos, porém, somente os metodos estáticos são considerados.

LCOM_3

Considera a coesão de métodos estáticos ou de instância. Calculada pela soma de LCOM com LCOM_2.

A ferramenta calcula os valores das métricas LCOM e LCOM_2 e retorna a soma dos resultados destas duas métricas.

Resposta para uma classe - RFC

A métrica *Response for a Class* (RFC) é calculada através da soma do número de métodos da classe mais os métodos que são invocados diretamente por eles. É o número de métodos que podem ser potencialmente executados em resposta a uma mensagem recebida por um objeto de uma classe ou por algum método da classe. Quando um método polimórfico é chamado para diferentes classes, cada diferente chamada identificada no bytecode é contada uma vez.

JaBUTi realiza para cada método da classe:

- monta o grafo GFC através do método **getGFC()**;
- para cada nó de GFC do tipo **GFCNode**, correspondentes aos comandos no programa, é verificado se o mesmo faz uma chamada à algum método, ou seja, se o tipo do nó no BG é (**GFCCallNode**);
- se fizer, uma chamada é acrescentada;
- a soma das chamadas à métodos de cada um dos métodos mais o número de métodos da classe é o valor final da métrica.

Observe que se determinado método faz várias chamadas para um mesmo método, a chamada deste método é contada apenas uma vez pela ferramenta.

Acoplamento entre objetos - CBO

Há acoplamento entre duas classes quando uma classe usa métodos e/ou variáveis de instância de outra classe. A métrica *Coupling Between Object* (CBO) é calculada através

da contagem do número de classes às quais uma classe está acoplada de alguma forma. O valor CBO de uma classe A equivale ao número de classes das quais a classe A utiliza algum método e/ou variável de instância, o que inclui o acoplamento baseado em herança, visto que um construtor da superclasse é sempre chamado no construtor de A.

JaBUTi através do método **getConstantPool()**, verifica os métodos e as variáveis de instância usadas na classe. Para cada método e variável de instância em **ConstantPool** é verificado o nome da classe a que pertence este método ou variável de instância. Os novos nomes de classe que surgem são armazenados em uma **HashSet**. O tamanho final da **HashSet** é o valor resultante da métrica.

4.3 OUTRAS MÉTRICAS

Complexidade Ciclomática de McCabe - CC

A métrica *Cyclomatic Complexity Metric* (CC) calcula a complexidade do método, através dos grafos de fluxo de controle que descreve a estrutura lógica do método. Os grafos de fluxo consistem de nós e ramos, onde os nós representam comandos ou expressões e os ramos representam a transferência de controle entre estes nós. A métrica pode ser calculada de várias formas, sendo uma delas: número de ramos - número de nós + 2. Os construtores são considerados. Mais esclarecimentos sobre a métrica podem ser vistos em [39] e [20]. São aplicadas mais duas variações para a métrica, além de WMC_CC vista acima.

CC_AVG

Valor médio dos valores da métrica CC de cada método da classe.

A métrica é calculada através do execução do método **CC()** em cada método da classe. Se o valor for menor que zero o método retorna o valor de -1. O valor retornado por cada um destes métodos é somado e dividido pelo número de métodos da classe, que dá o resultado da métrica.

CC_MAX

Valor máximo obtido dos valores da métrica CC entre os métodos da classe.

Também, calcula-se o valor CC de cada método da classe e comparando-se estes valores, o maior equivale ao resultado da métrica.

O próximo capítulo apresenta um estudo de caso, onde a Ferramenta JaBUTi é utilizada para coletar essas métricas a partir do código objeto de dois sistemas desenvolvidos em Java. Embora essa coleta tenha sido utilizada na tentativa de relacionar as métricas com a propensão a falhas das classes analisadas, seu objetivo maior não foi o de avaliar ou validar as métricas em si, mas apenas o de mostrar a factibilidade de coletá-las a partir do bytecode, ao invés do código fonte.

CAPÍTULO 5

ESTUDO DE CASO

As métricas implementadas na Ferramenta JaBUTi, descritas na seção anterior, foram utilizadas num estudo de caso utilizando dois sistemas. O primeiro, chamado μCode [38], considerado um sistema mais simples que o segundo, a própria Ferramenta JaBUTi, um sistema um tanto maior e mais complexo.

O estudo de caso procurou relacionar as métricas OO com a propensão a falhas, a exemplo do que foi feito em trabalhos encontrados na literatura como [16, 19, 33, 35, 37, 46]. Para tal, foram utilizadas duas versões de cada sistema, sendo uma a evolução da outra. Sobre a primeira versão (mais antiga) foram aplicadas as métricas OO através da Ferramenta JaBUTi. A segunda versão (atualizada) serviu como referência para que se pudesse avaliar em quais classes foram encontradas falhas. Foram consideradas "falhas" na versão original, todas as classes que sofreram alguma modificação de código na versão atualizada. Essa abordagem não é exata, pois uma modificação numa classe não indica necessariamente a existência de uma falha. Porém, com a inexistência de sistemas dos quais se consiga um registro de erros, decidiu-se adotar essa forma de avaliação para as classes com propensão a falhas.

Fez-se, então, a análise dos dados procurando relacionar o valor das métricas coletadas com a propensão a ocorrência de falha. Ressalta-se que o objetivo principal desse estudo de caso não foi realmente avaliar as métricas OO em relação à sua capacidade de indicar classes propensas à existência de falhas, mas a de mostrar a factibilidade que tais experimentos, como os realizados por [16, 19, 33, 35, 37, 46] possam ser realizados no nível de código objeto Java.

A técnica estatística utilizada para as análises dos dados foi a regressão logística (descrita na Subseção 2.5.1). Esta técnica baseia-se na construção de um modelo linear que é usado para explorar o relacionamento (caso exista) entre uma variável res-

posta binária (variável dependente) e uma ou mais variáveis independentes [6]. Da mesma forma que para os trabalhos anteriormente executados nesta área, descritos na Seção 2.5.2 [16, 19, 33, 35, 37, 46], foram construídos modelos de regressão logística simples (de apenas uma variável independente) para cada métrica, através dos quais foram selecionadas as métricas que fizeram parte do modelo de regressão logística múltiplo.

Pela análise simples, para cada variável independente (métrica), foi construído um modelo de regressão logística simples. De cada um destes modelos obtivemos os seguintes valores (discutido no Capítulo 2):

- Testes estatísticos Wald Chi-Square, Score e teste da máxima verosimilhança (*Likelihood Ratio* - G) e seus respectivos p-value associados para testar a hipótese nula de que todos os coeficientes de regressão são iguais a 0;
- Estimativas de máxima verosimilhança (coeficiente estimado, erro padrão, estatística Wald Chi-Square e seu p-value associado);
- Estimativa da razão de chance (*odds ratio*).

As variáveis resultantes da análise simples selecionadas para estarem presentes no modelo de regressão logística múltiplo foram as que apresentaram altos valores para as estatísticas G e Wald Chi-Square e nível de significância de 0.25 nos p-value associados a estas estatísticas, como comentado na Subseção 2.5.1.

Pela análise múltipla, o modelo de regressão logística múltiplo produz o mesmo conjunto de dados do modelo de regressão logística simples, com as seguintes diferenças:

- executa-se o método de seleção de variáveis (*Best Selection*) quando a análise simples (análise dos dados dos modelos de regressão logística simples) resultar em um grande número de variáveis possíveis;
- no caso do modelo múltiplo resultar em mais de uma variável independente, verifica-se a possibilidade de haver interações nas variáveis independentes do modelo.

Nas próximas seções serão descritos os resultados obtidos das análises estatísticas das métricas aplicadas ao μCode e ao JaBUTi, respectivamente. Na Seção 5.3 serão

apresentadas algumas das considerações finais do estudo de caso com os dois sistemas efetuado.

5.1 μCODE

A Ferramenta μCode [38] é uma ferramenta *freeware* disponível na Internet no site: http://mucode.sourceforge.net. Ela é uma API Java que dá suporte ao desenvolvimento de programas com código móvel. Ela é relativamente pequena e possui 16 classes e 4 versões liberadas. Para análise dos dados, foram utilizadas a versão (1.0), liberada em 7/9/2000 para aplicação das métricas e a versão atual (1.03), liberada em 1/8/2002. Das 16 classes que compõem o programa, 6 sofreram algum tipo de alteração da versão inicial para a versão atualizada.

Na Figura 5.1, pode-se visualizar a árvore de hierarquia de herança das 16 classes do Sistema μCode. As setas pontilhadas representam a hierarquia de interfaces.

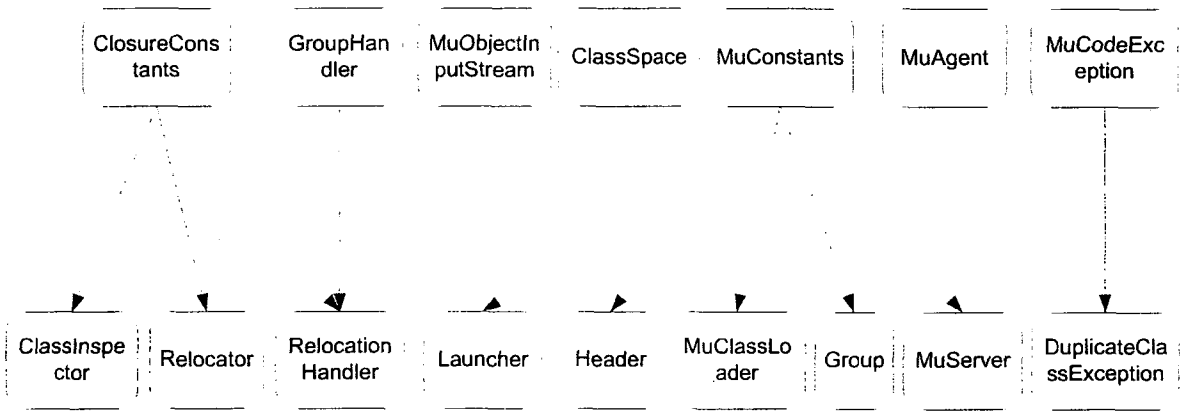


Figura 5.1: HIERARQUIA DE CLASSES DO μCODE

5.1.1 Análise Simples

Foram construídos modelos simples para cada uma das 27 métricas coletadas para as métricas resultantes da análise simples, conforme descrito anteriormente. O resumo dos resultados obtidos destes modelos simples pode ser visto na Tabela 5.1.

Resultados relevantes obtidos através da análise dos dados para este modelo:

Tabela 5.1: RESULTADOS DA ANÁLISE SIMPLES NO μCODE

Métrica	Coef. Estim.	Erro Padrão	Wald Chi-Square	p-value	$\Delta\Psi$
NMOS	0	.	.	.	1
NCV	-0.215	0.1865	1.3287	0.249	0.807
CC_MAX	0.0819	0.0841	0.9495	0.3298	1.085
NIV	0.9486	0.6214	2.3306	0.1269	1.880
ANPM	0.1784	0.5009	0.1268	0.7218	1.195
AMZ_SIZE	0.0147	0.0225	0.4296	0.5122	1.015
NCM_2	0	0.2130	0	1	1
NCM	-0.008	0.1141	0.0049	0.9441	0.992
NMAS	0	.	.	.	1
LCOM	0.1267	0.2113	0.3598	0.5486	1.135
NII	1.2953	0.8095	2.5604	0.1096	3.652
LCOM_3	0.0106	0.00763	1.9458	0.1630	1.011
AMZ_LOCM	0.0541	0.1003	0.2906	0.5899	1.056
LCOM_2	-0.0154	0.0281	0.3009	0.5833	0.985
NOC	-10.888	283.4	0.0015	0.9693	<0.001
DIT	-10.888	283.4	0.0015	0.9693	<0.001
RFC	0.0173	0.0115	2.2519	0.1334	1.017
WMC_SIZE	0.0026	0.00178	2.1132	0.1460	1.003
WMC	0.1489	0.0803	3.4352	0.0638	1.161
NPIM	0.2888	0.1550	3.4713	0.0624	1.323
CBO	0.1769	0.0977	3.2818	0.0701	1.194
WMC_CC	0.0214	0.0210	1.0453	0.3066	1.022
MNPM	0.3095	0.3586	0.7452	0.3880	1.363
SI	12.139	269.7	0.002	0.9641	>999.999
CC_AVG	0.4102	0.4796	0.7317	0.3923	1.509
NMIS	-1.5554	40.4795	0.0015	0.9693	0.211
WMC_LOCM	0.0108	0.0077	1.9369	0.1640	1.011

- somente as métricas NCV, NIV, NII, LCOM_3, RFC, WMC_SIZE, WMC, NPIM, CBO e WMC_LOCM foram consideradas indicadores significantes no nível de 0.25 do p-value associado à estatística Wald Chi-Square. Desta forma, 17 das 27 métricas calculadas foram descartadas na primeira etapa da análise dos dados;

- os valores para as métricas NMAS e NMOS não foram computados devido aos seus valores para as 16 classes do μCode [38] serem iguais a 0;

- Os baixos valores das métricas de herança DIT, NOC, NMIS, NMAS, NMOS, NII e SI indicam que neste sistema houve um uso insignificante de herança;

- A métrica NII (métrica de herança) demonstra que quanto maior o uso da herança, maior a tendência da ocorrência de falha no sistema ($\Delta\Psi$=3.652);

- Das medidas de tamanho a que mais influencia a tendência à falha é a métrica NIV

($\Delta\Psi$=1.880).

5.1.2 Análise Múltipla

Nesta etapa da análise dos dados foi criado um modelo com as 10 variáveis consideradas indicadores significantes à propensão a falhas, resultantes da análise simples. Com altos níveis de significância para o p-value associado à estatística Wald Chi-Square, os resultados demonstram que o modelo não ajusta corretamente o conjunto de dados.

Para encontrar o melhor modelo que ajustasse corretamente os dados do sistema, foi executado o método de seleção de variáveis *Best Selection*. A Tabela 5.2 mostra os resultados parciais da execução do método.

Tabela 5.2: RESULTADOS PARCIAIS DA EXECUÇÃO DO MÉTODO *BEST SELECTION* NO μCODE [38]

Número de Variáveis	Score Chi-Square	Variáveis Incluídas no Modelo
1	5.889	NPIM
2	7.149	NII NPIM
3	9.524	WMC_SIZE WMC_LOCM WMC_1
4	11.554	NII WMC_SIZE WMC_LOCM WMC_1
5	13.528	NII WMC_SIZE WMC_LOCM NPIM WMC_1

Segundo os resultados do método de seleção, foram construídos alguns modelos. Os modelos com 4 e 5 variáveis, não ajustaram bem este conjunto de dados, fato constatado através da observação do baixo valor da qualidade de ajuste e do valor alto do nível de significância do p-value associado à estatística Wald Chi-Square das métricas. Desta forma, constatou-se que para este conjunto de dados, quanto mais variáveis o modelo apresenta, menor é a sua capacidade de ajuste. Os resultados dos modelos construídos mais ajustados para este conjunto de dados podem ser vistos logo abaixo.

5.1.2.1 Modelo 1

O Modelo 1 possui duas variáveis independentes, a métrica de herança NII e a de tamanho NPIM. A qualidade de ajuste do modelo com o valor do *loglikelihood* é de 12.103. Os dados do modelo 1 podem ser vistos na Tabela 5.3.

Tabela 5.3: RESULTADOS DO MODELO 1 NO μCODE

Métrica	Coef. Estim.	Erro Padrão	Wald Chi-Square	p-value	$\Delta\Psi$
Intercept	3.5283	1.878	3.5298	0.0603	
NII	-1.4596	0.155	1.8165	0.1777	0.232
NPIM	-0.3515	0.2088	2.8344	0.0923	0.704

Resultados relevantes obtidos através da análise múltipla dos dados do Modelo 1:

- A razão de chance da métrica NPIM mostra-nos que um acréscimo de 1 unidade no valor de NPIM, aumenta em 0.704 a chance de encontrar-se uma falha na classe;
- O nível de significância do p-value associado à estatística Wald Chi-Square do modelo 1 de 0.20 é considerado bastante elevado para um modelo múltiplo. O valor mínimo selecionado que é considerado um bom ajuste para o modelo de dados é de 0.10 (discutido no Capítulo 2);
- O modelo foi aplicado para as 16 classes do Sistema μCode para comparar a propensão a falhas entre classes preditas e atuais. Uma classe foi classificada como predita a propensão a falhas, quando sua probabilidade de predição de conter uma falha for superior a 0.50. Este valor foi selecionado para aproximar o número de classes propensas a falhas preditas e atuais;
- O modelo identificou 10 classes como propensas a falhas, 6 das quais atualmente são, de fato, propensas a falhas (aproximadamente 60% de exatidão).

5.1.2.2 Modelo 2

O Modelo 2 possui três variáveis independentes, ou seja, três métricas de tamanho: WMC_SIZE, WMC_LOCM e WMC_1. A qualidade de ajuste do modelo é inferior que a do Modelo 1 com o valor do *loglikelihood* de 5.055. Os dados do Modelo 1 podem ser vistos na Tabela 5.4.

Resultados relevantes obtidos através da análise dos dados para o Modelo 2:

- Devido ao elevado valor da razão de chance da métrica WMC_LOCM, observa-se

Tabela 5.4: RESULTADOS DO MODELO 2 NO μCODE

Métrica	Coef. Estim.	Erro Padrão	Wald Chi-Square	p-value	$\Delta\Psi$
Intercept	4.0171	3.2426	1.5348	0.2154	
WMC_SIZE	-0.2651	0.2363	1.2583	0.2620	0.767
WMC_LOCM	1.3285	1.1984	1.2290	0.2676	3.775
WMC1	-2.0331	1.9466	1.0909	0.2963	0.131

uma maior propensão de encontrar falhas, com o acréscimo de uma unidade através deste modelo;

- O nível de significância do p-value associado à estatística Wald Chi-Square demonstra que este modelo não ajusta bem os dados do μCode;

- O modelo foi aplicado para as 16 classes do Sistema μCode com o intento de comparar a propensão a falhas entre classes preditas e atuais. Uma classe foi classificada como predita a propensão a falhas, se sua probabilidade de predição de conter uma falha for superior ou igual que 0.50. Este valor foi selecionado para aproximar o número de classes propensas a falhas preditas e atuais;

- O modelo identificou 11 classes como propensas a falhas, 6 das quais atualmente são propensas a falhas, demonstrando desta forma, uma exatidão de 54%.

5.1.2.3 Modelo 3

O Modelo 3 possui apenas uma variável independente, a métrica de tamanho, NPIM. A qualidade de ajuste do modelo é considerada a melhor dos modelos construídos com o valor do *loglikelihood* de 14.168. Os dados do Modelo 3 podem ser vistos na Tabela 5.5:

Tabela 5.5: RESULTADOS DO MODELO 3 NO μCODE

Métrica	Coef. Estim.	*Standard Error*	Wald Chi-Square	p-value	*odds ratio*
Intercept	-2.0112	0.9531	4.4531	0.0348	
NPIM	0.2798	0.1487	3.5403	0.0599	1.323

Os resultados relevantes obtidos através da análise dos dados para este modelo serão descritos a seguir:

- De acordo com o valor obtido na razão de chance da métrica deste modelo, deduz-se que a propensão de encontrar falhas no modelo diminui com o acréscimo de uma unidade;
- Através do nível de significância do p-value associado à estatística Wald Chi-Square, pode-se considerar este modelo como sendo o melhor para este conjunto de dados;
- O modelo foi aplicado para as 16 classes do Sistema μCode para comparar a propensão a falhas entre classes preditas e atuais. Uma classe foi classificada como predita a propensão a falhas, no caso de sua probabilidade de predição de conter uma falha for maior que 0.50. Este foi valor o selecionado para aproximar o número de classes propensas a falhas preditas e atuais;
- O modelo identificou 7 classes como propensas a falhas, 6 das quais são atualmente propensas a falhas (aproximadamente 86% de exatidão).

Através destes três modelos construídos observa-se que quanto maior o número de variáveis no modelo, eleva-se o nível de significância dos p-value associado à estatística Wald Chi-Square e consequentemente, diminui a qualidade de ajuste com menor exatidão do modelo. Levando-se em consideração que o modelo mais ajustado para os dados do μCode é o Modelo 3, que possui apenas uma variável independente (NPIM), não foi possível testar-se interações entre as métricas.

Em relação aos resultados apresentados por outros trabalhos semelhantes, pode-se notar:

- Como em [46], a métrica RFC demonstrou estar relacionada com a propensão de encontrar falhas, assim como CBO e WMC, mas com a diferença de que a métrica LCOM não mostrou ser insignificante para todos os casos;
- Em [16], os valores de NOC e DIT foram mínimos, o mesmo ocorrido com este estudo de caso;

- Como em [32], as métricas RFC, CBO, WMC foram encontradas pela análise simples associadas com propensão de encontrar falhas. A métrica NMAS foi considerada neste estudo de caso, uma métrica descartável pela análise simples;

- Como em [35] e [19], a métrica NOC e em [37] a métrica DIT não foram associadas com propensão a falhas.

5.2 JaBUTi

O segundo programa no qual as métricas foram aplicadas corresponde à própria Ferramenta JaBUTi. Ela possui 50 classes e algumas versões de desenvolvimento. Para análise dos dados, foram utilizadas duas versões, a primeira de 6/11/2002 e a segunda de 17/01/2003. Destas 50 classes, 13 sofreram algum tipo de alteração.

Na Figura 5.2, pode-se visualizar a árvore de hierarquia de herança das 50 classes do Sistema JaBUTi.

5.2.1 Análise Simples

Igualmente, foram construídos modelos simples para cada uma das 27 métricas coletadas para as métricas resultantes da análise simples. O resumo dos resultados obtidos destes modelos simples pode ser visto na Tabela 5.6.

Resultados relevantes obtidos através da análise dos dados para este modelo:

- apenas 9 métricas (NCV, NIV, AMZ_SIZE, LCOM_2, DIT, WMC_SIZE, MNPM, SI e WMC_LCOM) foram descartadas e consideradas indicadores insignificantes no nível de 0.25 do p-value associado à estatística Wald Chi-Square;

- da mesma forma que no Sistema μCode, também ocorreu neste sistema um uso insignificante de herança indicado pelos baixos valores das métricas SI, NII, NMOS, DIT e NOC. Para estas métricas de herança foram considerados valores baixos, até 2;

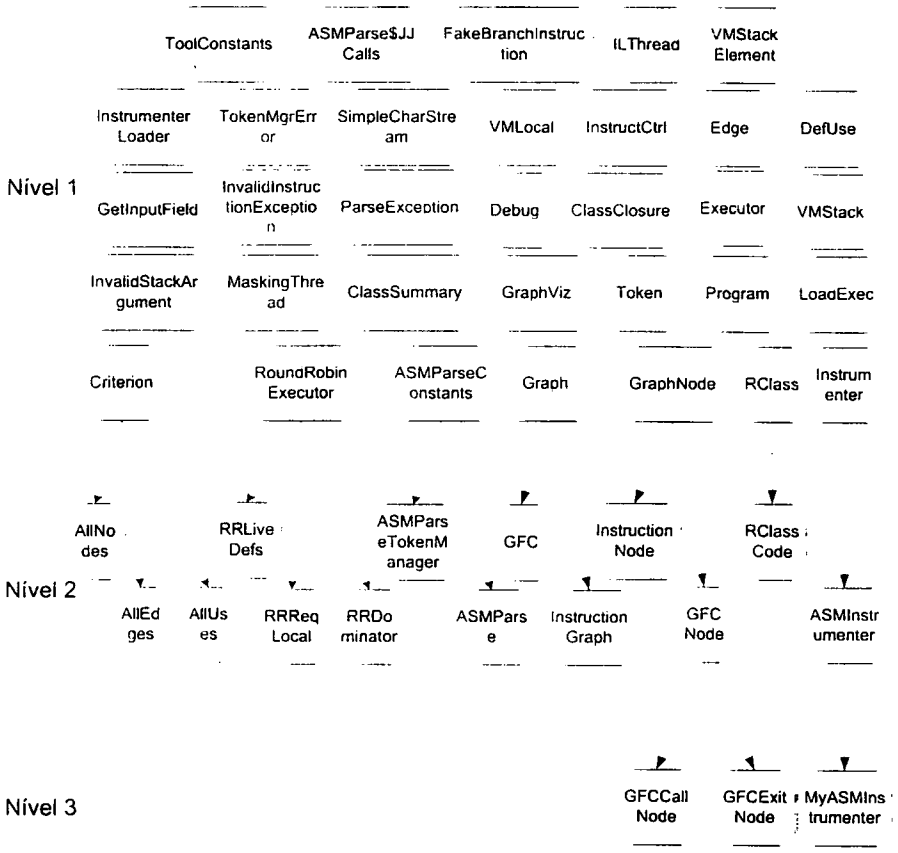


Figura 5.2: HIERARQUIA DE CLASSES DO JaBUTi

- Classes com elevado valor para as métricas CC_MAX, NMAS, LCOM, LCOM_3, AMZ_LOCM, NOC, RFC, CC_AVG, WMC_1, NPIM, CBO, WMC_CC, NMIS, sofreram alterações da primeira para a segunda versão do programa, como pode ser observado através dos resultados da análise simples. Desta forma, as métricas de complexidade, herança e algumas de tamanho se mostraram úteis na propensão de encontrar falhas neste sistema;

- As métricas NOC, NII e NMOS (métricas de herança) demonstram que quanto maior o uso da herança, maior a tendência da ocorrência de falha no sistema ($\Delta\Psi$=4.191; $\Delta\Psi$=2.844; $\Delta\Psi$=2.223);

- A maior parte das métricas de herança e de complexidade mostraram ser indicadores significantes na propensão à ocorrência de falhas neste sistema.

Tabela 5.6: RESULTADOS DA ANÁLISE SIMPLES NA FERRAMENTA JaBUTi

Métrica	Coef. Estim.	Erro Padrão	Wald Chi-Square	p-value	$\Delta\Psi$
NMOS	0.799	0.6344	1.5863	0.2079	2.223
NCV	-0.0043	0.0138	0.0954	0.7574	0.996
CC_MAX	0.0564	0.0301	3.5061	0.0611	1.058
NIV	0.0352	0.0553	0.406	0.524	1.036
ANPM	-0.6508	0.4845	1.8044	0.1792	0.522
AMZ_SIZE	-0.0004	0.00276	0.0252	0.874	1
NCM_2	0.4576	0.278	2.7095	0.0998	1.58
NCM	0.2361	0.1691	1.9503	0.1626	1.266
NMAS	0.4757	0.2286	4.3284	0.0375	1.609
LCOM	0.012	0.00791	2.3054	0.1289	1.012
NII	1.3863	0.7458	3.4549	0.0631	2.844
LCOM_3	0.0165	0.00981	2.8445	0.0917	1.017
AMZ_LOCM	0.0765	0.0427	3.2046	0.0734	1.079
LCOM_2	0.0557	0.0658	0.7168	0.3972	1.057
NOC	1.4329	0.7043	4.1391	0.0419	4.191
DIT	0.2818	0.5498	0.2628	0.6082	1.326
RFC	0.0179	0.00641	7.7728	0.0053	1.018
WMC_SIZE	0.0001	0.0002	0.193	0.6604	1
WMC_1	0.0622	0.0333	3.4926	0.0616	1.064
NPIM	0.0786	0.0446	3.1085	0.0779	1.082
CBO	0.0414	0.0283	2.1475	0.1428	1.042
WMC_CC	0.0192	0.0104	3.4014	0.0651	1.019
MNPM	-0.2122	0.245	0.7499	0.3865	0.809
SI	-0.858	1.0243	0.7015	0.4023	0.424
CC_AVG	0.4993	0.1962	6.4774	0.0109	1.648
NMIS	0.0635	0.0333	3.6252	0.0569	1.066
WMC_LOCM	0.0014	0.0014	0.9778	0.3228	1.001

5.2.2 Análise Múltipla

Nesta etapa da análise dos dados criou-se um modelo com as 18 variáveis consideradas indicadores significantes, resultantes da análise simples. Como no Sistema μCode, com elevados níveis de significância para o p-value associado à estatística Wald Chi-Square, os resultados demonstraram que o modelo não ajusta corretamente o conjunto de dados.

Com o intuito de encontrar o melhor modelo que ajuste corretamente os dados do sistema, executou-se o método de seleção de variáveis *Best Selection*. A Tabela 5.7 mostra os resultados parciais da execução do método.

Segundo os resultados do método de seleção, foram construídos alguns modelos. Os modelos com 6 e 7 variáveis, não demonstraram um bom ajuste para este conjunto de dados, devido à baixa qualidade de ajuste e do elevado nível de significância do p-value associado à estatística Wald Chi-Square das métricas. Desta forma, constatou-se que,

Tabela 5.7: RESULTADOS PARCIAIS DA EXECUÇÃO DO MÉTODO *BEST SELECTION* NO JaBUTi

Número de variáveis	Score Chi-Square	Variáveis Incluídas no Modelo
1	10.7280	RFC
2	15.9850	RFC ANPM
3	21.2093	CC_AVG LCOM_3 ANPM
4	25.2037	RFC CC_AVG NOC ANPM
5	26.1634	RFC CC_AVG NOC NII ANPM

para este conjunto de dados, os modelos que possuem um número superior a seis variáveis independentes apresentaram menor ajuste em relação aos modelos com um número menor de variáveis independentes. Os resultados dos 4 modelos melhor ajustados a este conjunto de dados podem ser vistos logo abaixo.

5.2.2.1 Modelo 1

O Modelo 1 possui duas variáveis independentes, as métricas de complexidade e de tamanho, RFC e ANPM, respectivamente. A qualidade de ajuste do modelo foi considerada favorável com o valor do *loglikelihood* equivalente a 40.642. Os dados do Modelo 1 podem ser vistos na Tabela 5.8.

Tabela 5.8: RESULTADOS DO MODELO 1 NO JaBUTi

Métrica	Coef. Estim.	Erro Padrão	Wald Chi-Square	p-value	$\Delta\Psi$
Intercept	1.3752	0.5895	5.4409	0.0197	
RFC	-0.0256	0.00847	9.1398	0.0025	0.975
ANPM	1.4764	0.6722	4.8241	0.0281	4.377

Resultados relevantes obtidos através da análise dos dados para o Modelo 1:

- Pela razão de chance da métrica ANPM, observa-se que, quanto maior o valor da métrica, maior é a propensão de encontrar uma falha na classe (o acréscimo de 1 unidade no valor de NOC, aumenta em 4.377 a chance de encontrar uma falha na classe):

- O nível de significância do p-value associado à estatística Wald Chi-Square de 0.05 do modelo demostrou que este modelo ajusta bem os dados do JaBUTi:

- O modelo foi aplicado para as 50 classes do Sistema JaBUTi a fim de comparar a propensão a falhas entre classes preditas e atuais. Uma classe é classificada como predita à propensão a falhas, caso sua probabilidade de predição de conter uma falha seja maior igual a 0.90. Este valor foi selecionado para aproximar o número de classes propensas a falhas preditas e atuais;

- O modelo identificou 19 classes como propensas a falhas, 13 das quais, através da análise estatística, foram identificadas como sendo propensas a falhas (68% de exatidão).

5.2.2.2 Modelo 2

O Modelo 2 possui três variáveis independentes, duas métricas de complexidade e uma de tamanho, LCOM_3, CC_AVG e ANPM, respectivamente. A qualidade de ajuste do modelo demonstra ser inferior à Modelo 1 com o valor do *loglikelihood* equivalente a 29.148. Os dados do Modelo 1 podem ser vistos na Tabela 5.9.

Tabela 5.9: RESULTADOS DO MODELO 2 NO JaBUTi

Métrica	Coef. Estim.	Erro Padrão	Wald Chi-Square	p-value	$\Delta\Psi$
Intercept	2.5404	0.9367	7.3553	0.0067	
CC_AVG	-1.1464	0.4097	7.8315	0.0051	0.318
LCOM_3	-0.0285	0.0148	3.6948	0.0546	0.972
ANPM	3.0123	1.0943	7.5777	0.0059	20.335

Resultados relevantes obtidos através da análise dos dados para o Modelo 2:

- Devido ao elevado valor da razão de chance da métrica ANPM observa-se um aumento expressivo na propensão de encontrar falhas com o acréscimo de uma unidade desta métrica;

- Da mesma forma, o nível de significância do p-value associado à estatística Wald Chi-Square do modelo foi considerado baixo e demonstrou que este modelo ajusta bem os dados do JaBUTi;

- O modelo foi aplicado para as 50 classes do Sistema JaBUTi para comparar a propensão a falhas entre classes preditas e atuais. Uma classe foi classificada como

predita a propensão a falhas, caso sua probabilidade de predição de conter uma falha seja maior igual que 0.90. Este valor foi selecionado para aproximar o número de classes propensas a falhas preditas e atuais;

- O modelo identificou 26 classes como propensas a falhas, das quais 13 foram identificadas como sendo propensas a falhas (50% de exatidão), através da análise estatística.

5.2.2.3 Modelo 3

O Modelo 3 possui quatro variáveis independentes, duas métricas de complexidade (RFC e CC_AVG), uma de herança (NOC) e uma de tamanho (ANPM). A qualidade de ajuste do modelo foi inferior às dos Modelos 1 e 2 com o *loglikelihood* equivalente a 26.647. Os dados do Modelo 3 podem ser vistos na Tabela 5.10.

Tabela 5.10: RESULTADOS DO MODELO 3 NO JaBUTi

Métrica	Coef. Estim.	Erro Padrão	Wald Chi-Square	p-value	$\Delta\Psi$
Intercept	-2.9856	0.9834	9.2170	0.0024	
RFC	0.0200	0.00955	4.3988	0.0360	1.020
CC_AVG	0.9666	0.3864	6.2579	0.0124	2.629
NOC	2.0884	1.0420	4.0168	0.0451	8.072
ANPM	-2.9358	1.1361	6.6775	0.0098	0.053

Resultados relevantes obtidos através da análise dos dados para o Modelo 3:

- O melhor modelo que ajustou adequadamente os dados do Jabuti foi o modelo que está esquematizado na Tabela 5.10;

- Pelo considerável valor da razão de chance da métrica NOC observa-se que, através deste modelo, o acréscimo de uma unidade na métrica aumenta consideravelmente a propensão de encontrar uma falha no sistema;

- O bom nível de significância do p-value associado à estatística Wald Chi-Square de 0.05 do modelo demonstrou que este modelo, também ajusta bem os dados do sistema;

- O modelo foi aplicado para as 50 classes do Sistema JaBUTi para comparar a propensão a falhas entre classes preditas e atuais. Uma classe foi classificada como predita a propensão a falhas, caso sua probabilidade de predição de conter uma falha é maior que 0.90. Este valor foi selecionado para aproximar o número de classes propensas a falhas preditas e atuais;

- O modelo identificou 26 classes como propensas a falhas, das quais 13 foram propensas a falhas, apresentando assim, 50% de exatidão.

5.2.2.4 Modelo 4

O Modelo 4 possui cinco variáveis independentes, duas métricas de complexidade (RFC e CC_AVG), duas de herança (NII e NOC) e uma de tamanho (ANPM). A qualidade de ajuste do modelo foi inferior que a do Modelo 3 com *loglikelihood* equivalente a 23.889. Os dados do Modelo 3 podem ser vistos na Tabela 5.11.

Tabela 5.11: RESULTADOS DO MODELO 4 NO JaBUTi

Métrica	Coef. Estim.	Erro Padrão	Wald Chi-Square	p-value	$\Delta\Psi$
Intercept	3.4664	1.1146	9.6719	0.0019	
RFC	-0.0163	0.00879	3.4242	0.0642	0.984
CC_AVG	-1.1121	0.4084	7.4144	0.0065	0.329
NOC	-2.2656	1.1717	3.7386	0.0532	0.104
NII	-2.2482	1.5057	2.2293	0.1354	0.106
ANPM	3.3181	1.2001	7.6444	0.0057	27.608

Resultados relevantes obtidos através da análise dos dados para o Modelo 4:

- Através do elevado valor da razão de chance da métrica ANPM observa-se novamente que ocorre o aumento à propensão de encontrar falhas com o acréscimo de uma unidade desta métrica;

- O elevado nível de significância do p-value associado à estatística Wald Chi-Square de 0.20 do modelo demonstrou que este modelo não ajusta de forma adequada os dados do JaBUTi;

- O modelo foi aplicado para as 50 classes do Sistema JaBUTi para comparar a propensão a falhas entre classes preditas e atuais. Uma classe foi classificada como

predita a propensão a falhas, se sua probabilidade de predição de conter uma falha é maior que 0.90. Este valor foi selecionado para aproximar o número de classes propensas a falhas preditas e atuais;

- O modelo identificou 28 classes como propensas a falhas, 13 das quais, através da análise estatística, foram identificadas como sendo propensas a falhas, equivalente à 46% de exatidão.

Através destes quatro modelos construídos observou-se que os três primeiros possuem um ótimo nível de significância do p-value associado à estatística Wald Chi-Square, boa qualidade de ajuste e exatidão, sendo que o Modelo 1 o que apresentou os melhores resultados. Através dos dados obtidos observou-se, que quanto maior for o número de variáveis no modelo, menor será a qualidade do ajuste do mesmo. Desta forma, os três primeiros modelos mostraram ajustar da melhor forma este conjunto de dados.

Considerando-se o Modelo 3, que apresentou o maior número de variáveis e demonstrou uma boa qualidade de ajuste aos dados do JaBUTi, procedeu-se tentativas de se incluir interações sobre as variáveis deste modelo. Os resultados desta inclusão são apresentados na Tabela 5.12:

Tabela 5.12: RESULTADOS DO MODELO 3 COM INTERAÇÕES NO JaBUTi

Métrica	Coef. Estim.	Erro Padrão	Wald Chi-Square	p-value	$\Delta\Psi$
RFC	-0.0220	0.0451	0.2391	0.6249	0.978
CC_AVG	-1.0155	1.4938	0.4621	0.4966	0.362
NOC	27.246	137.4	0.0393	0.8428	>999.999
ANPM	2.0364	1.4635	1.9363	0.1641	7.663
RFCxCC_AVG	-0.0047	0.0086	0.2955	0.5867	0.995
RFCxNOC	-0.4374	1.9809	0.0487	0.8253	0.646
RFCxANPM	0.0165	0.0660	0.0626	0.8024	1.017
CC_AVGxNOC	-1.9297	32.4738	0.0035	0.9526	0.145
CC_AVGxANPM	0.6191	1.3022	0.2260	0.6345	1.857
NOCxANPM	-6.0565	80.3673	0.0057	0.9399	0.002

Como as interações mostraram ser indicadores insignificantes à propensão a falhas, estas não foram inclusas no modelo. Assim, constatou-se que não houve interações entre as métricas do Modelo 3 para este conjunto de dados.

Na comparação com outros trabalhos pode-se notar que:

- Assim como em [46], as métricas NOC e RFC estavam relacionadas à propensão de encontrar falhas, mas contrariamente ao ocorrido neste estudo de caso, a métrica LCOM não foi insignificante em todos os casos;
- Em [16], os valores de NOC e DIT foram mínimos, como neste estudo de caso;
- Da mesma forma que em [32], pela análise simples, as métricas RFC, CBO, WMC e NMAS, também estavam associadas com a propensão de encontrar falhas;
- Como em [33], na análise simples, as métricas de herança e acoplamento, também mostraram-se fortemente relacionadas à probabilidade de encontrar falhas. Porém, as métricas de coesão não foram insignificantes;
- A métrica DIT não foi significante na associação com a propensão a falhas, como em [19].

5.3 CONSIDERAÇÕES FINAIS

Para o μCode, o modelo construído com apenas uma variável foi o que melhor ajustou o seu conjunto de dados. Além disso, através do resultado da execução do método de seleção de variáveis *Best Selection* verificou-se que as métricas selecionadas para fazerem parte do modelo com 3 variáveis passaram por muitas modificações, não contendo nenhuma das métricas do modelo com 1 e 2 variáveis. Estes fatos podem ser explicados devido ao reduzido número de observações (16 classes) em relação ao número de variáveis independentes calculadas no sistema (27 métricas).

Na análise dos dados do JaBUTi os resultados foram mais consistentes, tanto no que se refere aos resultados obtidos pela execução do método de seleção de variáveis *Best Selection*, quanto na medida *loglikelihood*, pois o aumento no número de variáveis no modelo diminuiu a qualidade de ajuste observada no mesmo. Os resultados da qualidade de ajuste, também foram coerentes com a exatidão observada nos modelos, pois uma maior qualidade de ajuste resultou em uma maior exatidão dos modelos quanto a observação das classes preditas à propensão a falhas e das atualmente identificadas como sendo falhas.

O baixo nível de significância associado à estatística Wald Chi-Square encontrado nos três primeiros modelos construídos no Sistema JaBUTi demonstrou que os mesmos ajustam de forma adequada o conjunto de dados do sistema e que estas métricas foram significantes na predição da ocorrência de falhas. Considerando-se que estes três primeiros modelos ajustam bem o conjunto de dados, o Modelo 3 com 4 variáveis foi considerado o que melhor ajusta o conjunto de dados do JaBUTi por conter o maior número de variáveis.

Através da extração dos dados do código objeto foi possível fazer a análise dos dados, obtendo resultados satisfatórios a respeito do relacionamento das métricas OO e de complexidade com a propensão de encontrar falhas em programas OO, da mesma forma que nos trabalhos executados anteriormente na área (Seção 2.5.2), nos quais foram utilizadas outras formas para se coletar estes dados, como por exemplo, o código fonte do programa e relatórios.

CAPÍTULO 6

CONCLUSÕES

Atualmente, com a necessidade de se obterem produtos de software de melhor qualidade e de se melhorar o processo de desenvolvimento do software, as métricas ganharam uma atenção diferenciada, uma vez que elas podem caracterizar erros de módulos do software trazendo esta melhora na qualidade do desenvolvimento do software e uma confiabilidade do produto final.

As métricas de software auxiliam na coleta de informações, fornecendo dados qualitativos e quantitativos sobre o processo e o produto de software. Elas identificam onde os recursos são necessários e são fontes cruciais de informações para a tomada de decisão resultando em uma economia de custos e recursos.

Com o surgimento do novo paradigma OO e sem a possibilidade de aproveitamento das métricas do paradigma estruturado, desconsiderando os novos conceitos introduzidos com esta nova tecnologia, como classe, polimorfismo, herança e encapsulamento, algumas métricas OO têm sido propostas.

Devido a esta necessidade de se trabalhar com novas métricas OO para sistemas deste paradigma, este trabalho propôs a aplicação das métricas OO mais importantes, que têm sido propostas na literatura de orientação a objetos, juntamente com a conhecida métrica do paradigma estruturado Complexidade Ciclomática de McCabe, em uma estrutura definida da representação do fluxo de controle e fluxo de dados de programas Java, gerada a partir do código objeto. Buscou-se validar empiricamente estas métricas no seu relacionamento com propensão à ocorrência de falhas nos módulos do software e verificar as implicações de coleta das métricas no código objeto e não no código fonte, através de dois sistemas, um de pequeno porte (μCode) e outro de médio porte (JaBUTi).

Os resultados gerados a partir dos ambientes dos estudos de casos descritos neste trabalho foram similares aos descritos na literatura para esta área, com a identificação de

métricas, como RFC, CBO e WMC, relacionadas à propensão de encontrar falhas nos módulos de software, em conjunto com outras métricas de LK. Como observado na Seção 2.5, os baixos valores para as métricas DIT e NOC demonstraram pouco uso de herança nos sistemas utilizados como estudos de casos, o qual também verificou-se com este. Porém, os resultados diferiram em alguns aspectos, devido às características peculiares de cada ambiente nos quais as métricas foram aplicadas. Ao observar que outros trabalhos relacionados à área apontam para resultados diferentes, quanto à aplicação de métricas no código fonte, aliada à compatibilidade de alguns resultados deste trabalho com aqueles, deduz-se que a aplicação de métricas no código objeto não acarreta em significantes implicações em relação ao código fonte, tornando-se, desta forma, uma ferramenta útil para o manejo do testador, em casos de indisponibilidade do código fonte.

O diferente enfoque quanto à aplicação de métricas no código objeto apresentado neste trabalho, demonstrou resultados satisfatórios através do estudo de caso desenvolvido, na tentativa de validar empiricamente as métricas OO e de avaliar a complexidade no seu relacionamento com a propensão às falhas, através da aplicação destas métricas no código objeto. Uma vez que resultados concretos e consideráveis foram alcançados, coletando-se os dados através da aplicação das métricas no código objeto, e comparando-os aos trabalhos nos quais os dados foram coletados do código fonte, pode-se concluir que a Ferramenta JaBUTi (Seção 3.2) contribui para a obtenção de qualidade do software, com a vantagem de poder ser utilizada nos casos em que o código fonte não está disponível.

BIBLIOGRAFIA

[1] *Information Technology - Software Product Quality - Part 1: Quality Model.* ISO/IEC FDIS 9126-1.

[2] H. Agrawal. Dominators, super block, and program coverage. *SIGPLAN - SIGACT Symposium on Principles of Programming Languages - POPL'94*, páginas 25–34, Portland, Oregon, janeiro de 1994. ACM Press.

[3] D. Azevedo, 1997. Disponível on-line: `http://www.pr.gov.br/celepar/celepar/batebyte/edicoes/1997/bb67/metrica%.htm`.

[4] A. Braga. *Análise de Pontos de Função.* Infobook, Rio de Janeiro, Brasil, 1996.

[5] L. B. Chaves. *Uma Avaliação Empírica de Métricas para Programas Orientados a Objeto no contexto de Teste de Software.* Tese de Doutorado, Universidade Federal do Paraná, Paraná, Brasil, 2001.

[6] D. Collet. *Modelling Binary Data.* Chapman & Hall, 1994.

[7] T. DeMarco. *Controlling Software Projects: Management, Measurement and Estimation.* Yourdon Press, New Jersey, 1982.

[8] R. Selby e A. Porter. Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Transactions On Software Engineering*, 14(12):1743–1757, Dezembro de 1988.

[9] M. I. Haufe e A. Price, 1999. Disponível on-line: `http://www.inf.ufrgs.br/pos/SemanaAcademica/Semana99/mariaisabel/mariai%sabel.html`.

[10] F. Brito e Abreu e W. Melo. Evaluating the impact of object-oriented design on software quality. *Proc. Third Int'l Software Metrics Symp.*, páginas 90–99, Berlin, Alemanha, Março de 1996.

[11] S. R. Chidamber e C. F. Kemerer. A metric suite for object oriented design. *IEEE Transactions On Software Engineering*, 20(6):467–493, 1994.

[12] S. Rapps e E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions On Software Engineering*, 11(4):367–375, Abril de 1985.

[13] T. Lindholm e F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2 edition, 1999.

[14] F. Lanubile e G. Visaggio. Evaluating predictive quality models derived from software measures: Lesson learned. Relatório Técnico CS-TR-3606, Universidade de Maryland, 1996.

[15] M. Lorenz e J. Kidd. *Object-Oriented Software Metrics - A Practical Guide*. Prentice Hall, 1994.

[16] M. Cartwright e M. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions On Software Engineering*, 26(8):786–796, 2000.

[17] W. Li. e S. Henry. Object-oriented metrics that predict maintainability. *Journal Systems and Software*, 23(2):111–122, Fevereiro de 1993.

[18] D. W. Hosmer e S. Lemeshow. *Applied Logistic Regression*. Wiley-Interscience, 1989.

[19] A. Binkley e S. Schach. Validation of the coupling dependency metric as a predictor of run-time faulures and maintenance measures. *Proc. 20th Int'l Conf. Software Engineering*, páginas 452–455, Osaka, Japão, Abril de 1998.

[20] A. H. Watson e T. J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric, 1996.

[21] J. Munson e T. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions On Software Engineering*, 18(5):423–433, Maio de 1992.

[22] P. Nesi e T. Querci. Effort estimation and prediction of object-oriented systems. *Journal Systems and Software*, 12(1):89–102, Julho de 1998.

[23] S. D. Conte e V. Y. Shen. *Software Engineering Metrics and Models.* Menlo Park, 1985.

[24] S. Benlarbi e W. Melo. Polymorphism measures for early risk prediction. *Proc. 21st Int'l Conf. Software Eng.*, páginas 334–344. 1999.

[25] R. Gunning. *Techniques of Clear Writing.* Nova York: McGraw-Hill, 1962.

[26] M. H. Halstead. *Elements of Software Science.* Elsevier Computer Science Library, Nova York, USA, 1977.

[27] W. Harrison. Using software metrics to allocate testing resources. *J. Management Information Systems.* 4(4):93–105. Abril de 1988.

[28] W. Harrison. Software measurement: A decision-process approach. *Advances in Computers*, 39:51–105, 1994.

[29] International Function Point User Group. *Análise por Pontos por Função*, 1991. Baseado na Release 3.4 do Manual de Práticas de Contagem do IFPUG".

[30] International Function Point User Group. *Function Point Counting Practices Manual: V4.0 Atlanta*, 1994.

[31] C. Jones. *Programming Productivity.* McGraw-Hill, 1986.

[32] N. Goel e S. N. Rai K. Emam, S. Benlarbi. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions On Software Engineering*, 27(7), Julho de 2001.

[33] J. W. Daly e V. Porter L. C. Briand, J. Wüst. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal Systems and Software*, 51:245–273, 2000.

[34] P. Devanbu e W. Melo L. C. Briand. An investigation into coupling measures for c++. *Proc. 19th Int'l Conf. Software Eng. - ICSE*, páginas 412–421, Boston, USA, 1997.

[35] S. Ikonomovski e H. Lounis L. C. Briand. J. Wüst. A comprehensive investigation of quality factors in object-oriented designs: An industrial case study. Relatório Técnico ISERN-98-29. Int'l Software Eng. Research Network, 1998.

[36] V. R. Basili e C. Hetmanski L. C. Briand. Developing interpretable models with optimized set reduction for identifying high risk software components. *IEEE Transactions On Software Engineering*, 19(11):1028–1044, Novembro de 1993.

[37] M-H Kao e M-H Chen M-H Tang. An empirical study on object oriented metrics. *Proc. Sixth Int'l Software Metrics Symp.*, páginas 242–249, 1999.

[38] G. P. Picco. μCode: A Lightweight and Flexible Mobile Code Toolkit. *Proc. 2nd International Workshop on Mobile Agents 98 (MA'98)*. páginas 160–171. Stuttgart. Alemanha. Setembro de 1998.

[39] R. S. Pressman. *Engenharia de Software.* Makron Books. 1 edition. 1997.

[40] R. S. Pressman. *Engenharia de Software - Uma Aproximação para Praticantes.* McGraw-Hill, 5 edition, 2000.

[41] S. Counsell e R. Nithi R. Harrison. An overview of object-oriented design metrics. *8th UK International Workshop on Software Technology and Engineering Practice (STEP '97)*, Londres, Inglaterra, Julho de 1997.

[42] S. Counsell e R. Nithi R. Harrison. Coupling metrics for object oriented design. *Proc. Fifth Int'l Symp. Software Metrics*, páginas 150–157, Bethesda, Maryland. Março de 1998.

[43] N. E. Fenton e B. Kitchenham S. L. Pfleeger. Toward a framework for software measurement validation. *IEEE Transactions On Software Engineering*, 23(3):187–188, Março de 1997.

[44] D. Darcy e C. F. Kemerer S. R. Chidamber. Managerial use of metrics for object oriented software: An exploratory analysis. *IEEE Transactions On Software Engineering*, 24(8):629–639, Agosto de 1998.

[45] A. S. Panday e H. B. More T. M. Khohgoftaar. A neural network approach for predicting software development faults. *Proc. Third Int'l IEEE Symp. Software Reliability Engineering*, Carolina do Norte, 1992.

[46] L. C. Briand e W. L. Melo V. R. Basili. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions On Software Engineering*, 22(10):751–761, Outubro de 1996.

[47] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, e W. E. Wong. Jaba: A java bytecode analyzer. *Proc. Simpósio Brasileiro de Engenharia de Software - Sessão de Ferramentas*, Gramado, RS, Outubro de 2002.

[48] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, e W. E. Wong. Jabuti - java bytecode understanding and testing - user's guide - version 1.0. Relatório técnico, ICMC/USP, 2003. (em preparação).

[49] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, e W. E. Wong. JaBUTi: A coverage analysis tool for Java programs. *XVII SBES – Simpósio Brasileiro de Engenharia de Software*, Manaus, AM, Brasil, outubro de 2003. (submetido para a Sessão de Ferramentas).

[50] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, e W. E. Wong. Java bytecode static analysis: Deriving structural testing requirements. *2nd UK Software Testing Workshop UK-Softest'2003*, páginas –, Department of Computer Science, University of York, York, England, setembro de 2003. Universidade de York Press. (aceito para publicação).